编程狂人

Programming Madman

NO.1

推酷

关于推酷

推酷是专注于 IT 圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等多方面内容，满足你日常的专业阅读需要。我们针对 IT 人还做了个活动频道，它聚合了 IT 圈最新最全的线上线下活动，使 IT 人能更方便地找到感兴趣的活动信息。

关于周刊

推酷周刊是专为 IT 人打造的行业技术周刊，目前推出的《编程狂人》是献给广大的程序员们。 我们利用技术挖掘出那些高质量的文章，并通过人工加以筛选整理出来。 每期的周刊一般会在周六到周一的某个时间点发布。关于《编程狂人》的名号还有个小插曲， 因为我们一直很崇尚程序员这个群体，所以我们一开始将周刊定名为《编程匠人》， 不过做封面设计的朋友不知怎么就改成了《编程狂人》，我们想想也还不错，就使用了。关于《编程狂人》， 如果你有意见和建议，欢迎反馈给我们。

联系我们


tuicool2012


164644910


推酷网

下载 APP

Android版本

iPhone版本

目录

业界新闻

前端开发

编程语言

HashMap 深度解析(一)

Java 中使用内存映射文件需要考虑的 10 个问题

目录

Facebook 运维内幕曝光 :一人管理 2 万台服务器

目前，Facebook 已经凭借它在网络基础建设上的可扩展能力成为了行业的领军者。Facebook 数据中心运维主管 Delfina Eberly（下图人物） 在“7x24 Exchange 2013 秋季会议”上的演讲中为我们透露了 Facebook 部分内部运维数据，下面我们来具体了解下。

Facebook 数据中心运维主管 Delfina Eberly

服务器数量惊人，一人管理 2 万台

Facebook 服务器数量惊人，其硬件方面的工作重点主要放在“可服务性”上，内容也涉及服务器的初期设计，一系列工作的目标就是为了保证数据机房的设备维修最简单、最省时。她介绍说，每个 Facebook 数据中心的运维工作人员管理了至少 20,000 台服务器，其中部分员工会管理数量高达 26,000 多个的系统。

近期 Facebook 的服务器与管理人数比又创下了新高，目前已经超过 10000:1，

可以查看文章高扩展性对此进行更加详细的了解。

大数据汹涌，运维工作不轻松

在 Facebook 数据中心做运维工作并不轻松，对工作人员的能力要求很高。他们每天面对的是海量数据。

据统计，Facebook 目前拥有 11.5 亿用户，日常登录用户约 7.2 亿。每天 Facebook 用户分享的内容达到 47.5 亿条，“赞”按钮点击次数近 45 亿次。Facebook 目前存储了 2400 亿张照片，每月照片存储容量约增加 7 PB（注，单位换算：1PB=1024TB）。

自动故障诊断系统：原为留住人才

为了管理运维工作，Facebook 已经开发了相应软件来自动化处理日常运维任务，如 CYBORG 可自动检测服务器问题并进行修复。如果 CYBORG 无法自动修复检查出的问题，系统将自动给订单系统发送警告，并分派给数据中心工作人员，以对相应问题进行详细追踪与分析。

Eberly 提到，自动化工作的目标是尽量避免将技术人员派往现场解决问题，除非必须对服务器进行现场处理。强调自动化不是因为 Facebook 对打造无人数据中心感兴趣，原因在于 Facebook 重视自己的员工。

Eberly 解释说：我们要留住人才，因为大家更喜欢高水平的任务，公司希望让他们留下来与我们一起进步成长，这对 Facebook 来说至关重要。

“可服务性”主导服务器设计：节时 54%

在 Facebook，运维团队的时间与工作量是根据 Facebook 硬件设计来安排的。比方说，全部服务器从头开始就坚持“可服务性”这一原则来进行设计，那么数据中心的工作人员就没有必要老钻机房了；服务器被设计成无需工具就可以对磁盘和组件进行替换。这样做的结果就是：Facebook 用来修理服务器的时间减少

了 54%。

Eberly 介绍说，Facebook 运维团队会仔细跟踪设备故障率，这一数据会为公司的采购提供参考。公司的财产管理和订单系统用序列号来跟踪硬盘和其他组件，这方便完整了解每个硬件的生命周期。

Eberly 还提到，虽然这些系统很复杂，但并不需要太多开发者。Facebook 的运维团队仅有 3 名软件工程师，但他们对数据中心的工作来讲至关重要。

最后

从 Eberly 的介绍中，我们可以看到 Facebook 在可扩展性网络建设上的实力。同时，这也为行业提供了一些可参考的经验，如：开发自动故障系统，根据“可服务性”设计基础架构。同时，运维也是一个系统工程，需要得到其他部门的配合支持才行。

原文：

http://www.iteye.com/news/28480-facebook-ops-staffer-manages-20000-servers

谷歌深度学习系统超预期 人类已无法经理解电脑想法

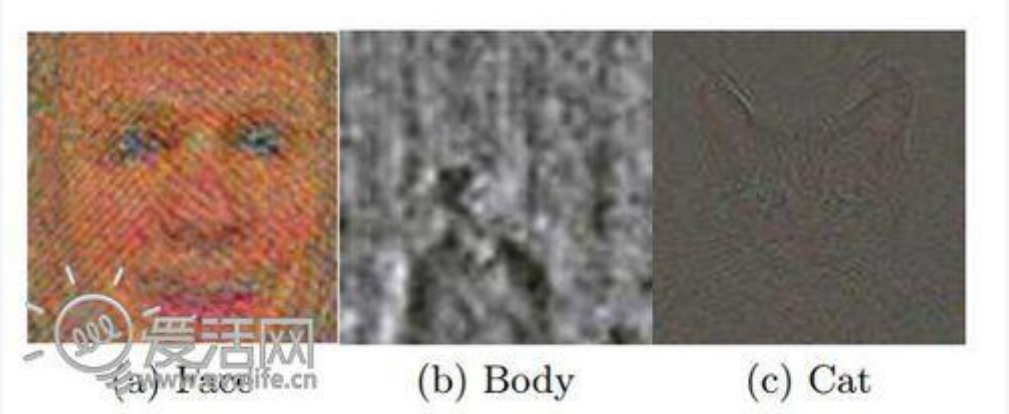
虽然科幻电影描绘人工智能已经到机器能够独立思考的程度，但在现实生活中，受限于硬件设备的处理能力和编程逻辑的复杂性，我们身边的的人工智能仍然显得比较幼稚和容易理解——毕竟人类创造了所谓的人工智能，它们不可能超乎人类所能理解的范畴发展。

但谷歌的“深度学习（deep learning）”系统却颠覆了这一常识，谷歌的工程师都表示**这套原本只是用来做实验的决策计算机系统表现超乎想象，它通过照片识别事物的能力早已超乎了谷歌工程师的预料，谷歌工程师甚至已经不知道计算机究竟在“想”些什么了。**

识别系统的基础

谷歌软件工程师 Quac V. Le 在上周五旧金山的机器学习大会（Machine Learning Conference）上谈到了这套深度学习系统。这是一套包含了大量服务器群，能够收集并自动对数据作归类的系统，谷歌打算用它来深度研究 AI 技术。在谷歌手中，此系统的服务应用包括了 Android 的语音控制搜索、图片识别和谷歌翻译。

深度学习系统曾在去年 6 月份引起过不少讨论，当时纽约时报刊文称谷歌的 DistBelief 技术（一个采用普通服务器的深度学习并行计算平台）在获取数百万 YouTube 视频数据后，能够精准地识别出这些视频的关键元素：猫。未来，这套系统或是能够准确识别谷歌街景照片中门牌号码、网站中人脸图片等的技术依托。



深度学习技术理论上也是分层结构。其神经网络的最底层可检测到图片像素在色彩上的变化，上层随后可了解图片中出现特定事物的边缘部分。位于再上层的几个连续的分析层可通过系统的不同分支学会人脸、摇椅、计算机等各种类型事物的检测方法。

它们真的在“独立思考”

Quoc V. Le 说，令他最为震惊的事情是，**深度学习系统能够轻易地学习总结出类似碎纸机等物体的特性，这些甚至是普通人类难以轻易做到的。**“怎样在系统设计中让软件能够具备识别碎纸机的能力，这是相当复杂的。我在这方面花了很多的时间，但就是难以完成。”

实际上 Quoc 也曾给身边的好多朋友看了碎纸机的照片，但在随后的识别过程中，具有高等智慧的人类却遭遇了麻烦。而谷歌的深度学习计算机系统则在这方面具有极高的识别成功率，可关键问题是，Quoc 自己也不知道他所写的程序是如何做到这一点的。

也就是说，**谷歌的工程师们已经无法解释这套系统识别事物的方法和逻辑**，它们更像是脱离了其创建者的控制在独立思考，这种复杂的认知方式更是令人不可思议。虽然这种层面的“独立思考”范围还非常有限，但在实际应用中却真实有效，能够解决实际问题。谷歌负责 AI 研究的主管 Peter Norvig 认为这种能够

实现大量数据统计的模型对于解决如语音识别与理解一类的复杂问题具有非常积极的意义。

结论

Quoc 说，对谷歌而言，深度学习系统能够解决人类所不能解决的问题，自然也就是节约人力成本的好东西。将其更多的潜力挖掘出来，总好过雇佣一批每年拿着无数酬劳的高级专家。“机器学习非常复杂，我们需要花大量的时间在数据处理和特性更新上。甚至为了解决一个独立的问题，我们就需要聘请这一领域的专家。以后我们期望能够跳脱这样的模式，我们没法解决的问题，就让机器去完成。”

而且谷歌实际也在开发其他类似的决策选择系统，如 Borg 与 Omega，这些系统在分配工作负荷时，行为方式也更像是活物。将来，机器的“独立思考”或将真正成为可能。至少现在，我们让计算机与人类达成了这样的协作关系。

原文：http://www.evolife.cn/html/2013/74117.html

Redis 2.8.0 正式版发布

NoSQL Redis 2.8.0 正式版发布！新的产品线。2013-11-22 高性能 KV 数据库。经过 6 个 RC. 上个版本是 2013-08-29 的 2.6.16.

对比 2.6 所有新特性如下：

* [NEW] Slaves are now able to partially resynchronize with the master, so most
    of the times a full resynchronization with the RDB creation in the master
    side is not needed when the master-slave link is disconnected for a short
    amount of time.
* [NEW] Experimental IPv6 support.
* [NEW] Slaves explicitly ping masters now, a master is able to detect a timed out
    slave independently.
* [NEW] Masters can stop accepting writes if not enough slaves with a given
    maximum latency are connected.
* [NEW] Keyspace changes notifications via Pub/Sub.
* [NEW] CONFIG SET maxclients is now available.
* [NEW] Ability to bind multiple IP addresses.
* [NEW] Set process names so that you can recognize, in the "ps" command output,
    the listening port of an instance, or if it is a saving child.
* [NEW] Automatic memory check on crash.
* [NEW] CONFIG REWRITE is able to materialize the changes in the configuration
    operated using CONFIG SET into the redis.conf file.
* [NEW] More NetBSD friendly code base.
* [NEW] PUBSUB command for Pub/Sub introspection capabilities.
* [NEW] EVALSHA can now be replicated as such, without requiring to be expanded
    to a full EVAL for the replication link.

* [NEW] Better Lua scripts error reporting.
* [NEW] SDIFF performance improved.
* [NEW] Slaves are now able to partially resynchronize with the master, so most

    of the times a full resynchronization with the RDB creation in the master

    side is not needed when the master-slave link is disconnected for a short

    amount of time.

* [NEW] Experimental IPv6 support.
* [NEW] Slaves explicitly ping masters now, a master is able to detect a timed out

    slave independently.

* [NEW] Masters can stop accepting writes if not enough slaves with a given

    maximum latency are connected.

* [NEW] Keyspace changes notifications via Pub/Sub.
* [NEW] CONFIG SET maxclients is now available.
* [NEW] Ability to bind multiple IP addresses.
* [NEW] Set process names so that you can recognize, in the "ps" command output,

    the listening port of an instance, or if it is a saving child.

* [NEW] Automatic memory check on crash.
* [NEW] CONFIG REWRITE is able to materialize the changes in the configuration

    operated using CONFIG SET into the redis.conf file.

* [NEW] More NetBSD friendly code base.
* [NEW] PUBSUB command for Pub/Sub introspection capabilities.
* [NEW] EVALSHA can now be replicated as such, without requiring to be expanded

    to a full EVAL for the replication link.

* [NEW] Better Lua scripts error reporting.
* [NEW] SDIFF performance improved.
* [NEW] The new inline protocol now accepts quoted strings like, for example

    you can now type in a telnet session: set 'foo bar' "hello world\n".

* [NEW] Add per-db average TTL information in INFO output.
* [NEW] redis-benchmark improvements.
* [NEW] dict.c API wrong usage detection.
* [NEW] DEBUG SDSLEN for sds memory debugging.
* [NEW] SCAN, SSCAN, HSCAN, ZSCAN commands.
* [NEW] Log the new master when SLAVEOF command is used.
* [NEW] Sentinel code synchronized with the unstable branch, the new Sentinel

is a reimplementation that uses more reliable algorithms.

Redis 是一个高性能的 key-value 数据库。 redis 的出现，很大程度补偿了

memcached 这类 keyvalue 存储的不足，在部 分场合可以对关系数据库起到很好的补充作用。它提供了 Python，Ruby，Erlang，PHP 客户端，使用很方便。

性能测试结果：

SET 操作每秒钟 110000 次，GET 操作每秒钟 81000 次，服务器配置如下：

Linux 2.6, Xeon X3320 2.5Ghz.

stackoverflow 网站使用 Redis 做为缓存服务器。

完整的发布声明：
https://raw.github.com/antirez/redis/2.8/00-RELEASENOTES

下载：http://download.redis.io/releases/redis-2.8.0.tar.gz

Redis 的详细介绍：请点这里

Redis 的下载地址：请点这里

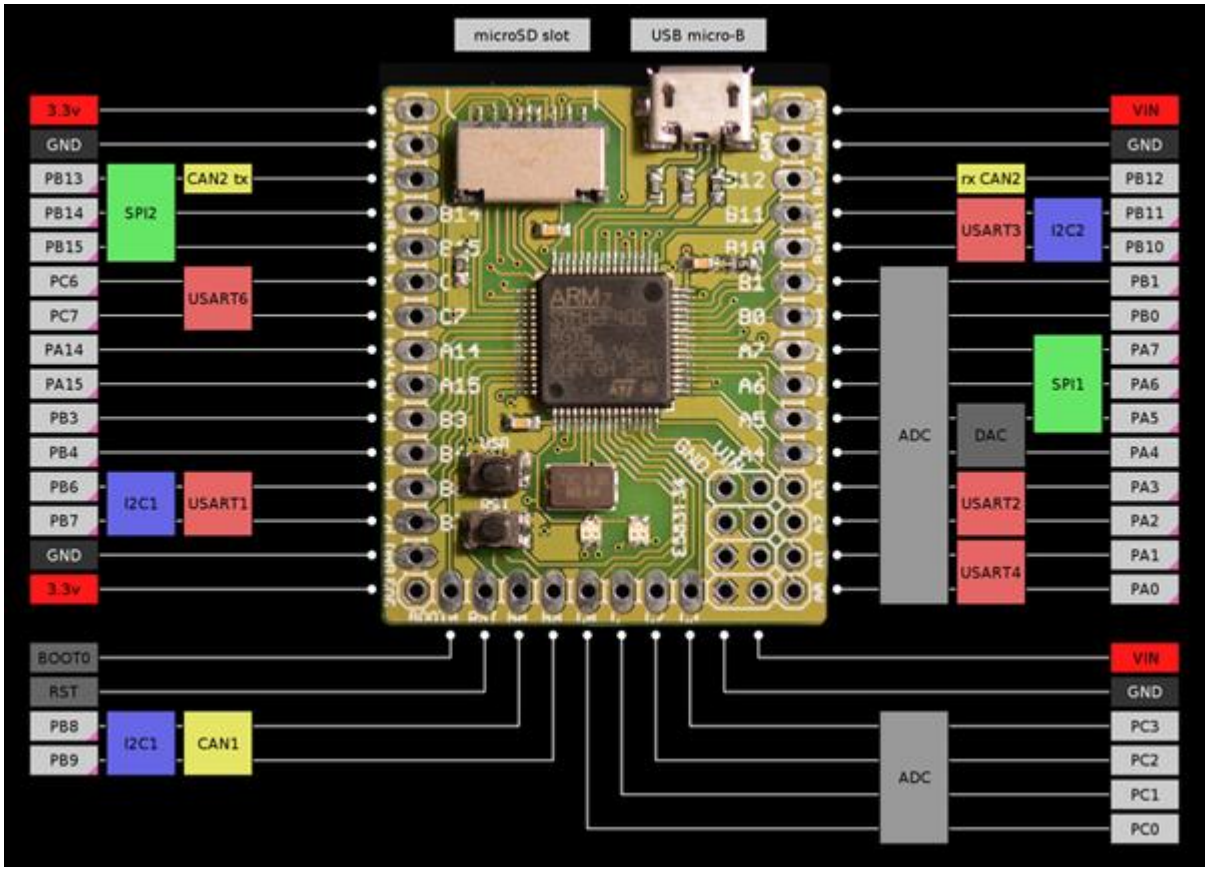
原文：http://www.oschina.net/news/46181/redis-2-8-0-final

Micro Python：用 Python 语言控制单片机

Arudino 虽然在开源硬件领域非常流行，但是对于不懂 C/C++编程语言的玩家来说，还是需要一段学习时间。

早些时候，我们在 Kickstarter 上看到一款相对特殊的开发板 Espruino，他能让用户通过 JavaScrpit 轻松实现对单片机的控制，让更多的计算机初学者来动手做硬件。

从现在开始，Python 玩家也可以做类似的事情。

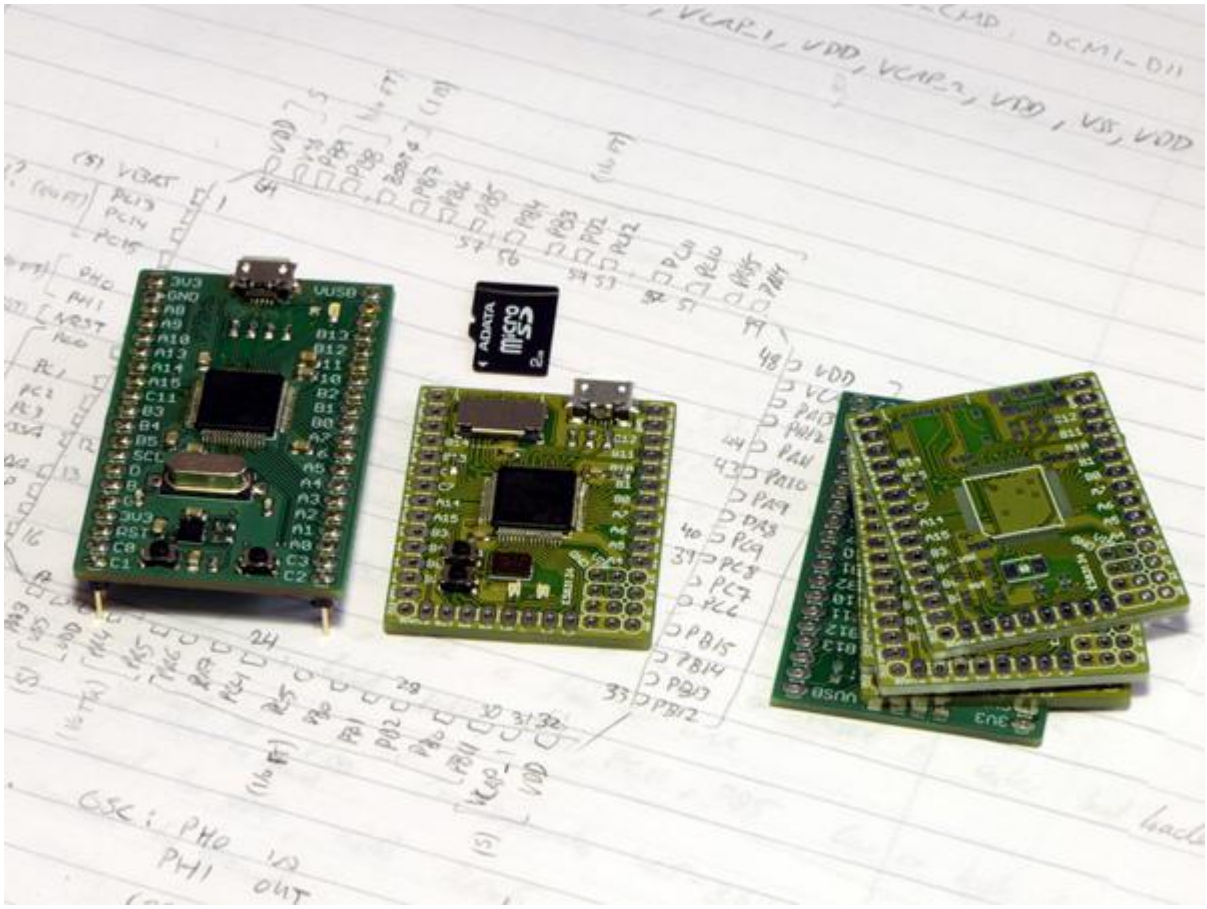


Damien George 是一名计算机工程师，他每天都要使用 Python 语言工作，同时也在做一些机器人项目。有一天，他突然冒出了一个想法：能否用 Python 语言来控制单片机，进行实现对机器人的操控呢？

要知道，Python 是一款比较容易上手的脚本语言，而且有强大的社区支持，一些非计算机专业领域的人都选它作为入门语言。遗憾的是，它不能实现一些非常底层的操控，所以在硬件领域并不起眼。

Damien 为了突破这种限制，他花费了六个月的时间来打造 Micro Python。它基于 ANSI C，语法跟 Pyton 3 基本一致，拥有自家的解析器、编译器、虚拟机和类库等。目前他支持基于 32-bit 的 ARM 处理器，比如说 STM32F405。

借助 Micro Python，用户完全可以通过 Python 脚本语言实现硬件底层的访问和控制，比如说控制 LED 灯泡、LCD 显示器、读取电压、控制电机、访问 SD 卡等。



与此同时，Damien 还给大家带来了一款专门为 Micro Python 而打造的开发板，它基于 STM32F405 单片机，通过 USB 接口进行数据传输。该开发板内置 4 个 LED 灯、一个加速传感器、时钟模块，可在 3V-10V 之间的电压正常工作。值得一提的是，它遵守 MIT 协议开源，被授权人拥有复制、修改、发行和再授权的权利。

这款板子的面积为 33mm×40mm，重 6 克。对它感兴趣的朋友可以上 Kickstarter 支持一下，最低售价为 20 英镑。

话说回来，未来是否会有更多的编程语言加入单片机领域呢？下一个会是 Java，还是 Go？

视频：http://v.youku.com/v_show/id_XNjM4MDI1Nzk2.html

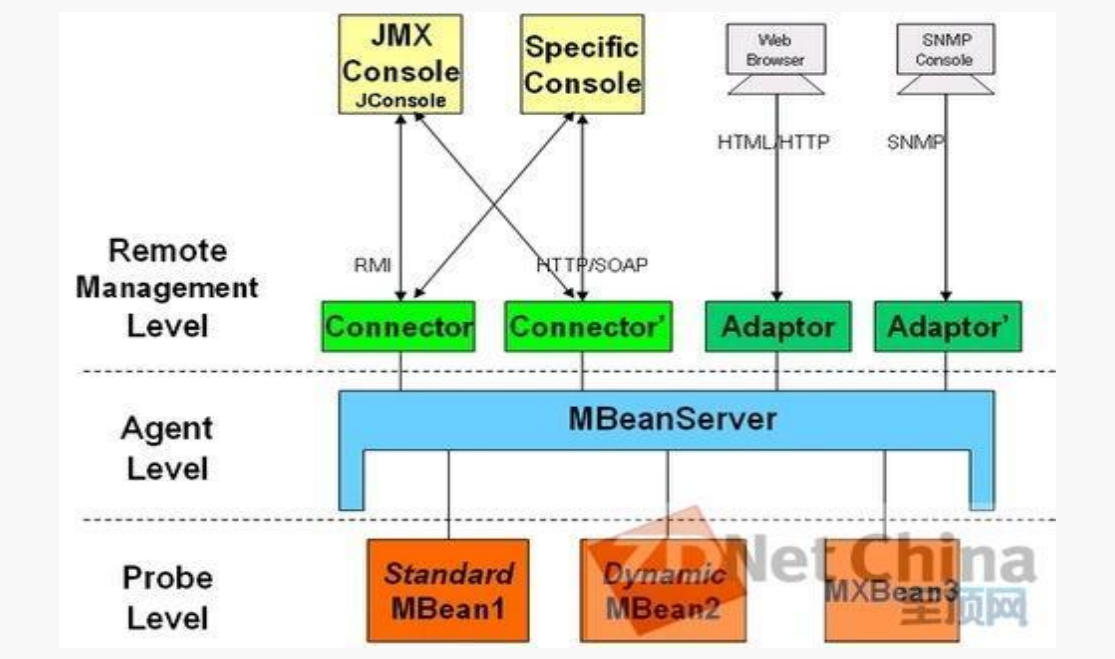
原文：http://www.leiphone.com/python-microcontroller.html

遭 JBoss 漏洞破坏 23000 台服务器“中招”

黑客利用曝光的 JBoss 管理界面和调用程序在服务器上安装 Web Shells。 攻击者正积极利用一个已知漏洞破坏 JBoss Java EE 应用服务器，这些应用服务器以非安全方式向互联网曝露了 HTTP 调用服务。

攻击者正积极利用一个已知漏洞破坏 JBoss Java EE 应用服务器，这些应用服务器以非安全方式向互联网曝露了 HTTP 调用服务。

十月初，安全研究员 Andrea Mlcalizzi 在多家厂商(包括惠普，McAfee，赛门铁克和 IBM)使用 4.X 和 5.X JBoss 的产品中发现了一个漏洞并将之发布。黑客可以利用该漏洞(CVE-2013-4810)在曝露了 EJBInvokerServlet 或 JMXInvokerServlet 上 JBoss 部署上安装一个任意应用。



Micalizzi 利用该漏洞安装了一个名改为 pwn.jsp 的 Web Shell 应用，该应用可通过 HTTP 请求在操作系统上执行 Shell 命令。可以通过 OS 的用户身份许可从而运行 JBoss，而在一些 JBoss 部署案例中，甚至可以拥有较高的权限，如管理员。

来自安全公司 Imperva 的研究员最近检测到针对 JBoss 服务器的攻击有所增加，这些攻击利用 Micalizzi 所说的漏洞安装了原始的 pwn.jsp shell，不仅如此还有更复杂的名为 JspSpy 的 Web Shell。

在 JBoss 服务器上运行的 200 多个站点,包括那些隶属于政府和大学的站点，都被这些 Web Shell 应用入侵和感染，Imperva 安全策略总监 Barry Shteiman 说。

实际情况更为严重，因为 Micalizzi 所说的漏洞源自不安全的默认配置，这些配置使得 JBoss 管理界面和调用程序暴露在未经验证的攻击之下,这一问题已经存在多年了。

2011 年，在一份关于 JBoss 因安装不当被黑的报告中，Matasano Security 的安全研究员在谷歌搜索的基础上估计约有 7300 台有潜在漏洞服务器。

据 Shteiman 透露，管理界面暴露到互联网的 JBoss 数量翻了三倍，达到了 23000。

这种增长的原因之一或许是人们完全了解与此问题相关的风险，不过却一直以不安全的方式部署安装 JBoss，Shteiman 说。而且，有些厂商推出产品的时候，本身就使用了不安全的 JBoss 配置，如不能抵挡 Micalizzi 漏洞利用的产品，他说。

存在 CVE-2013-4810 漏洞的产品包括 McAfee Web Reporter 5.2.1，惠普 ProCurve Manager 3.20 和 4.0，惠普 PCM+ 3.20 和 4.0，惠普 Identity Driven Manager 4.0，Symantec Workspace Streaming 7.5.0.493 和 IBM TRIRiGA。 还未发现有其他厂商的产品上有此漏洞。

JBoss 由 Red Hat 开发，最近更名为 WildFly。最新的版本是 7.1.1，不过据 Shteiman 透露，许多企业因兼容性的问题而仍使用 JBoss 4.X 和 5.X 版本，因为他们要运行的应用是为旧版的 JBoss 而研发。

这些企业应该到 JBoss 社区网站查询确保 JBoss 的安全安装。

IBM 也对此漏洞做出响应，提供了安全安装 JMX Console 和 EJBInvoker 的信息。

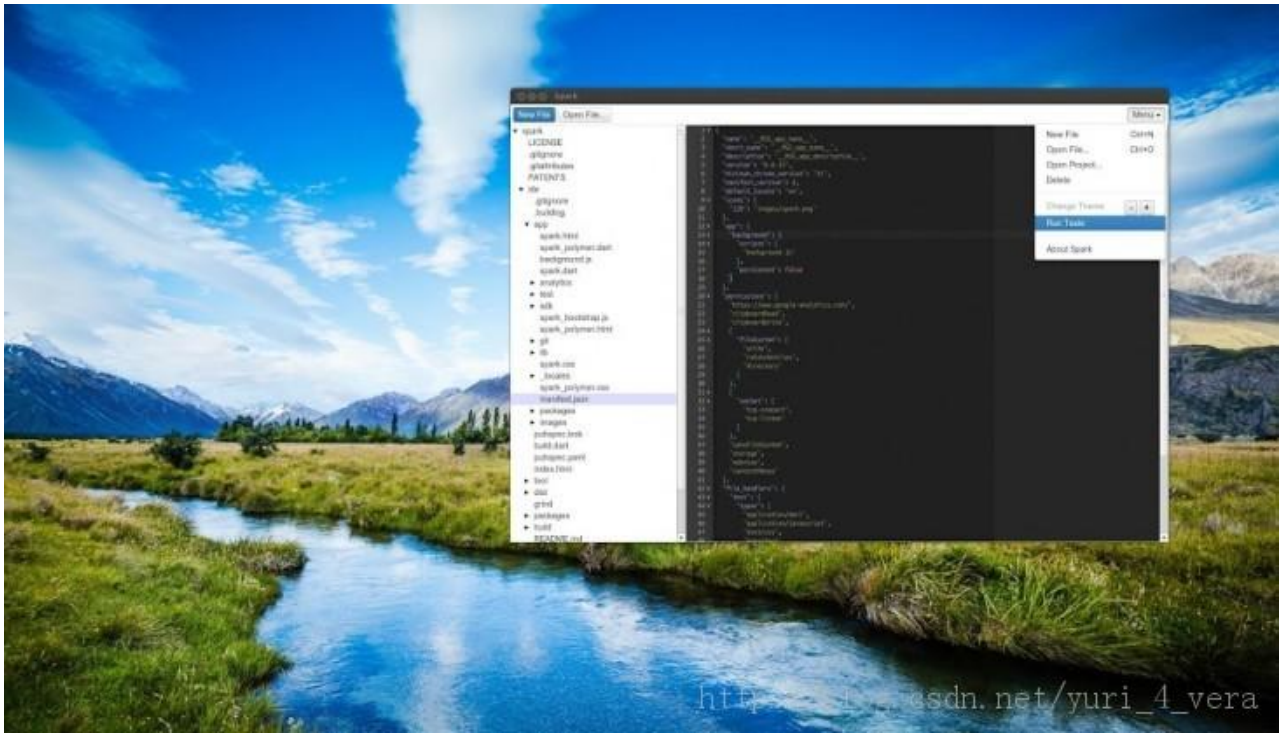
原文：http://netsecurity.51cto.com/art/201311/418130.htm

谷歌正在建立一个叫做 Spark 的 Chrome 应用开发环境

谷歌的 Chromium 团队一直让大家惊叹，他们最新的项目是一个叫做 Spark 的 Chrome 应用集成开发环境（IDE）。
这个新的应用最早被谷歌开源 Chromium 项目的布道师和开发者 François Beaufort 记录，这是他对新的 IDE 项目的描述：

- 它使用了被称为“用于可升级的网页应用开发新语言”的 Dart 语言构建，
- 它包含了由 Polymer 驱动的图形工具库。
- 它在 GitHub 上开源，因此任何感兴趣的人都可以了解到 Dart 和 Polymer 是怎样被用于建立新一代的 Chrome 应用。

以下是目前的进展：



也许你不知道，Chrome 中的应用是用 HTML,JavaScript 和 CSS 写的，但默认离线运行在
浏览器之外，并且能得到 Web 应用之外的特定的 API，从另一角度来讲，这是谷歌推动 Web 的极限使其成为平台的方式。
同时，Dart 也是谷歌的开源 Web 编程语言，它的基本目标就是取代 JavaScript。Polymer 是谷歌的 Web 库，建立于 Web 元素之上，旨在充分利用不断发展的网络平台上的现代浏览器。
目前尚不清楚谷歌是否会真正支持这个 Chrome 应用程序，并当它完成之后定期更新它。或者这个应用只会被单一的用于展示上述技术所具备的可能性。我们倾向于前者，但对于谷歌多久能完成项目，你永远也不知道。
你可以在这里和这里分别找到 IDE 和用户接口组件。显然还有很多工作要做，但再次强调，这是一项艰巨的任务。
原文：http://blog.csdn.net/yuri_4_vera/article/details/16897215

快速提升 CSS 编码能力的五个必备知识点

作为 web 开发人员，无论你是前端开发或者是后端开发，也无论你是新手还是大神，对于 CSS 的知识了解都是需要，在这篇文章中，我们将从五个不同的角度帮助大家快速的提升自己的 CSS 水平,如果你能够在以下五点有所突破的话,CSS 的编码能力肯定会有一定的提升。

1. 了解 CSS 中的定位（position）

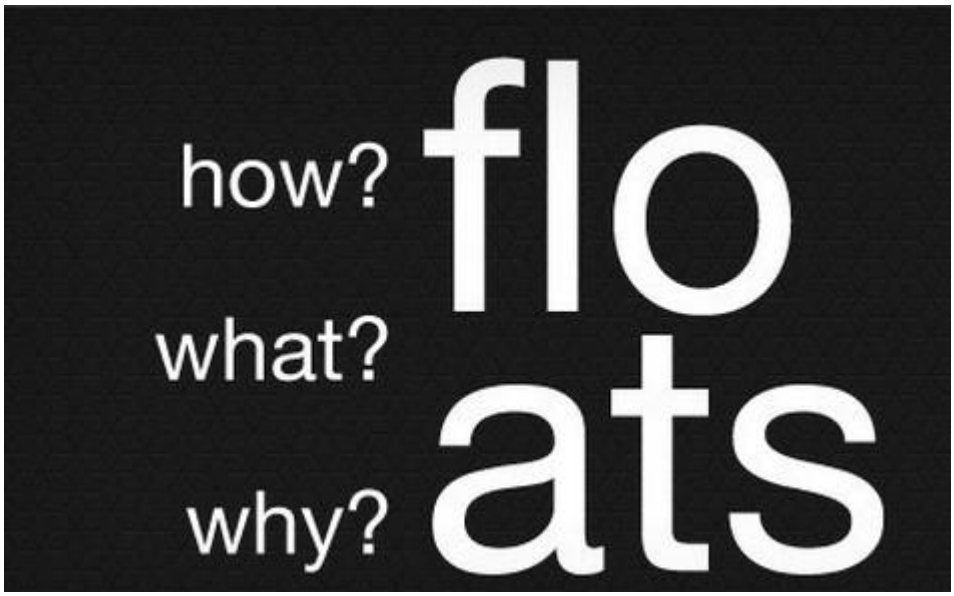
如果你需要深度的了解 CSS 如何移动 HTML 元素，或者控制元素的位置和显示的话，定位是一个不可获取的基本知识，而且除了了解简单和具有普遍性的定位外，你需要了解他们具体的不同。

其实这里包含了 5 个基本的定位方式，如果你不能直接说出名字来的话，那么你需要好好的了解一下这方便的知识了，包含：

- static
- relative
- absolute
- inherit
- fixed

以上五种定位你都需要了解，但是实际最常用的是 relative 和 absolue。

2. 熟练的掌握 float



当你初次学习 CSS 的时候，float 肯定是最让你头疼的地方，但是一旦你了解了基本的知识，你就慢慢会了解如何处理 clear fix 和 overflow 相关的操作。

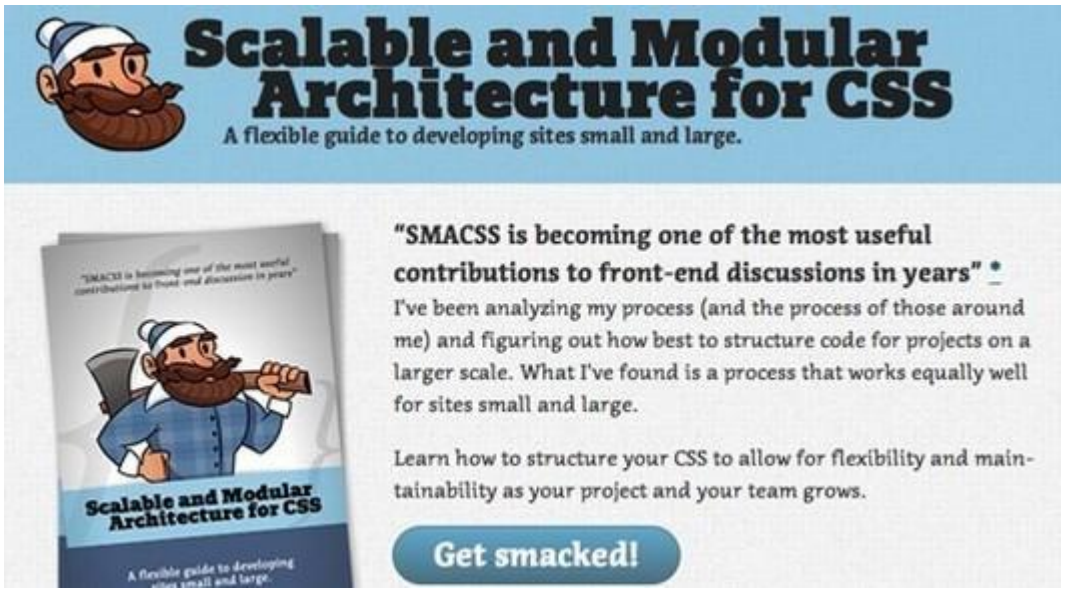
你需要的就是比较有深度的介绍 float 相关的技巧及其行为的文章。

3. 了解选择器

为了写出清爽并且简洁的 CSS，你需要对于选择有一定的了解，听起来可能比较简单，但是在实际的运用场景中，会比较复杂。

对于这个主题，需要学习的内容很多，虽然你可能觉得不需要了解那么多炫酷的选择器，但是对于每天都需要接触到 CSS 的我们来说，还是应该能力了解相关的含义。

4. 学习 DRY Coding 概念



“不要重复自己”，这个简单的俗语拥有非常深层次的含义。但你真正的开始使用 DRY coding 的时候，你将得到非常干净整齐的代码。

和前端我们介绍的主题不太一样，这个主题内容比较狭窄，主要就是使用 CSS 预编译。

当然很多人号称使用预编译导致了更差的代码编写，但实际相反。CSS 预编辑器帮助避免了重复的工作和内容。看看类似 LESS 和 SASS 的输出代码，可以看到它们让我写出了更加优美的代码。一旦你喜欢上了这样方式，你肯定会爱不释手的。

5. 了解浏览器的支持

最后一条就是你应该知道你的 CSS 在哪里可以正常工作。CSS3 可能无法忽视，但是可悲的现实是有些浏览器它并不支持（例如 IE 老版本）

对于一些新的开发工程师来说，可能并不是去记住那个 CSS 特性在浏览器中被支持或者不被支持，而是应该去哪里去查找。

原文： http://blog.yunshipei.com/?p=275

IE11 支持 JavaScript 新型 API，Web 应用国际化更便捷

摘要： IE 一度被认为是 HTML5 技术发展的绊脚石，而 IE11 正在打破这种局面，开发者无需担心前端效果无法在 IE 中运行，用户也可以获得和其他浏览器一致的体验。本文将介绍 IE11 对于 JavaScript 中最新的国际化 API 的支持情况。

IE 一度被认为是 HTML5 技术发展的绊脚石，而国内众多的 IE 6/7/8 用户令广大的前端开发者头痛不已，尽管 IE 9/10 对 HTML5 的支持有所改善，但还是不尽完美。微软也将希望寄托于 IE11 上。

据微软透露，IE11 目前已经完美支持目前的 HTML5 新特性，而在对 JavaScript 执行方面，性能也大幅提升，且支持最新的 API。开发者无需担心精心准备的前端效果无法在 IE 中运行，用户也可以获得和其他现代浏览器一致的浏览体验。

本文将为你介绍 IE11 对 JavaScipt 新的国际化特性方面的支持。

支持 ECMAScript 国际化 API

ECMAScript 国际化 API 提供了一个标准的 JavaScript 国际化接口，比如数字、日期、时间、货币格式以及特定文化的字符串，便于你的应用程序国际化。

在 IE11 中，Web 应用程序可以利用 Windows 的国际化库，其中包括支持超过 364 种语言环境、18 种数字系统、各种日期格式、各种日历系统等。

特定文化的字符串排序

IE11 可以自动实现特定区域设置字符串排序背后通常很复杂的逻辑。在不同的语言和文化间，字符串的排序和顺序惯例通常大相径庭。排序顺序可能会基于大小写敏感性、语音或者字符的可视化表示。例如，在东亚语言中，字符是按笔画和会意字的偏旁来排序。不同的语言和文化其排序还取决于字母排序顺序。例如，瑞典语有一个字符 "Æ"，排在 "Z" 后。德语也有字符 "Æ"，但是其排序像 "ae"，排在 "A" 后。

在 IE11 中，可以使用 Intl.Collator 构造函数，以及所需的区域设置和选项，来构造能区分不同区域性的排序器对象。排序器对象的比较方法可以用于比较两个字符串。String.prototype.localeCompare 已更新为可以内部使用 Intl.Collator，以便实现区分区域设置的比较，现在可以支持另外两个可选参数，区域设置和选项。

下面这个例子演示了如何对“Apple”、“Æble”、“Zebra”字符串进行排序。

```
1   var arr = ["Apple", "Æble", "Zebra"];
2   // Create collator object to use culture rules for English in the U.S.
3   var co = new Intl.Collator("en-US");
4   // Sorting array 'arr'  produces [Æble, Apple, Zebra] based on en-US rules
```

```
5   arr.sort(function(a, b) {
6           return co.compare(a, b);
7   });
8   // Create collator object to use culture rules for Danish in Denmark
9   var co = new Intl.Collator("da-DK");
10  // Sorting array 'arr' produces [Apple, Zebra, Æble] based on da-DK rules
11  arr.sort(function(a, b) {
12          return co.compare(a, b);
13  });
```

数字格式

E11 支持多种区域性和编号系统所用的惯例，用以设置数字的格式和显示形式。当表示和比较数字时，IE11 支持不同的格式，例如，“小数点”、“百分比”和“货币”。对于货币，显示选项包括“代码”和“符号”。不同的区域设置可以设置自己要显示的最小或最大整数、分数或有效数字。例如，对于区域设置"en-US"，小数 10000.50 将显示为 10,000.50 这样的格式，对于"de-DE"，该数字则显示为 10.000,50。

可以使用 Intl.NumberFormat 构造函数以及所需的区域设置标记和选项来构造区分区域性的 NumberFormat 对象。NumberFormat 对象的 format 方法可以用于基于区域设置和选项集来指定数字数据的格式。
Number.prototype.toLocaleString 已经更新为可以内部使用 Intl.NumberFormat，以提供区分区域性的格式。

您可以试用 World Data Test Drive，它使用 JavaScript 中提供的新数字格式 API 为数字指定格式，例如，小数、百分比或货币（包括本地货币显示）。

日期和时间格式

与各种不同的数字格式相类似，不同的地区日期和时间格式也差别很大，IE11 支持多个选项，诸如时区、年代、年份、月、工作日、日、小时、分钟和秒等等。

可以使用 Intl.DateTimeFormat 构造函数以及所需的区域设置标记和选项来构造区分区域性的 DateTimeFormat 对象。DateTimeFormat 对象的 format 方法可以用于基于区域设置和选项集为时间值指定格式。

在 World Data Test Drive 中，当单击并选择国家/地区时，您可以设置不同区域设置的最新人口普查日期格式，可以选择短月份和短年份等选项。

详细信息：Building world-ready applications in JavaScript using IE11

IE11 中的新功能：http://msdn.microsoft.com/library/ie/bg125382

体验 IE11：http://www.ietestdrive.com/

IE11 下载：http://msdn.microsoft.com/library/ie/aa740471

原文：
http://www.csdn.net/article/2013-11-20/2817568-IE-JavaScript-Internationalization

Qcon 和 CtripUED 上的分享：实时与可插入 WEB

除了自己办的 W3CTECH 会议，今年是希望少参加点会议，在多做点实物之后，再出去听出去讲。因此这一年改版了百姓网的前端框架 Puerh，写了 Pen 编辑器，做了 iNews.io 这个 HackerNews 类似的开源程序，和七牛合作做了 Staticfile 这个公益项目（目前每天有 30G 的新请求量，大约等下一天帮助开发者省下 100w 的 jQuery 新请求量），做了不少公司网站的优化，最后自己辞职，开始了创业生涯。

视频地址：sofish.de/file/realtime-web-technolgy/

处在这个 Mobile Web 盛行，Web 平稳发展的年代，这两个会议我选择了同样的一个话题 —— 实时与可插入 WEB。演讲的视频估计后面 infoQ 和 Ctrip UED 都会放出来，这里放一个纯 HTML 版的。点击显示下一页（请用 Safari 打开，可以播放视频）。

原文：http://sofish.de/2286

iOS 系列译文：Mach-O 可执行文件

当我们在 Xcode 中构建一个程序的时候，其中有一部分就是把源文件（.m 和.h）文件转变成可执行文件。这个可执行文件包含了将会在 CPU（iOS 设备上的 arm 处理器或者你 mac 上的 Intel 处理器）运行的字节
当我们在 Xcode 中构建一个程序的时候，其中有一部分就是把源文件（.m 和.h）文件转变成可执行文件。这个可执行文件包含了将会在 CPU（iOS 设备上的 arm 处理器或者你 mac 上的 Intel 处理器）运行的字节码。

我们将会过一遍编译器这个过程的做了些什么，同时也看一下可执行文件的内部到底是怎样的。其实，里面的东西比你看到的要多很多。

让我们先把 Xcode 放一边，踏入 Commond-Lines 的大陆。当我们在 Xcode 中构建一个 App 时，Xcode 只是简单的调用了一系列的工具而已。希望这将会让你更好的明白一个可执行文件（被称之为 Mach-O 可执行文件），是怎样组装起来的，并且是怎样在 iOS 或者 os x 上执行的

XCrun

先从一些基础性的东西开始：我们将会使用一个叫做 Xcrun 的命令行工具。他看起来很奇怪，但是的确相当出色。这个小工具是用来调用其他工具的。 原先的时候我们执行：

```
1.  % clang -v
```

现在在终端中，我们可以执行：

```
1.  % xcrun clang -v
```

Xcrun 定位 Clang，并且使用相关的参数来执行 Clang。

为什么我们要做这个事情？这看起来毫无重点，胡扯八道。但是 Xcrun 允许我们使用多个版本的 Xcode，或者使用特定 Xcode 版本里面的工具，或者针对特点的 SDK 使用不同的工具。如果你恰好有 Xcode4.5 和 xcode5、使用 xcode-select 和 xcrun 你可以选择选择使用来自 Xcode4.5 里面的 SDK 的工具，或者来自 Xcode5 里面的 SDK 的工具。在大多数其他平台上，这将是一个不可能的事情。如果你看一下帮助手册上 xcode-select 和 xcrun 的一些细节。你就能在不安装命令行工具的情况下，使用在终端中使用开发者工具。

一个不使用 IDE 的 Hello World

回到终端，创建一个包含一个 c 文件的目录：

```
% mkdir ~/Desktop/objcio-command-line
% cd !$
% touch helloworld.c
```

现在使用你喜欢的文本编辑器来编辑这个文件，例如 TextEdit.app：

```
% open -e helloworld.c
```

录入下面的代码：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("Hello World!\n");
    return 0;
}
```

保存，并且回到终端执行：

```
% xcrun clang helloworld.c
% ./a.out
```

现在你能够在终端上看到熟悉的 Hello World!。你编译了一个 C 程序并且执行了它。所有都是在不使用 IDE 的情况下做的。深呼吸一下，高兴高兴。我们在这里做了些什么？我们将 hellowrold.c 编译成了叫 a.out 的 Mach-o 二进制文件。a.out 是编译器的默认名字，除非你指定一个别的。

Hello World 和编译器

现在可选择的编译器是 Clang（读作：/’ kl /)。Chris 写了一些更多关于 Clang 细节的介绍，可以参考： about the compiler

概括一下就是，编译器将会读入处理 hellowrold.c，输出可执行文件 a.out。这

个过程包含了非常多的步骤。我们所要做的就是正确的执行它们。

预处理:
序列化
宏定义展开
#include 展开（引用文件展开）

语法和语义分析：
使用预处理后的单词构建词法树
执行语义分析生成语法树
输出 AST （Abstract Syntax Tree）

代码生成和优化
将 AST 转化成更低级的中间码（LLVM IR）
优化生成代码
目标代码生成
输出汇编代码

汇编程序
将汇编代码转化成目标文件
连接器

将多个目标文件合并成可执行文件（或者一个动态库） 我们来看一个关于这些步骤的简单的例子。

预处理
编译器将做的第一件事情是处理文件。使用 Clang 展示一下这个过程：

```
1.  % xcrun clang -E helloworld.c
```

欧耶。输出了 413 行内容。打开个编辑器看看到底发生了什么：

```
1.  % xcrun clang -E helloworld.c | open -f
```

在文件顶部我们能看到很多以”#”开头的行。这些被称之为行标记语句的语句告诉我们它后面的内容来自哪里。我们需要这个。如果我再看一下 hellowrold.c，第一行是：

```
1.  #include <stdio.h>
```

我们都用过#include 和#import。它们做的就是告诉于处理器在#include 语句的地方插入 stdio.h 的内容。在刚刚的文件里就是插入了一个以#开头的行标记。

跟在#后面的数字是在源文件中的行号。每一行最后的数字是在新文件中的行号。回到刚才打开的文件，接下来是系统头文件，或者一些被看成包裹着 extern “C” 的文件。

如果你滚动到文件末尾，你将会发现我们的 helloworld.c 的代码：

```
# 2 "helloworld.c" 2
int main(int argc, char *argv[])
{
        printf("Hello World!\n");
        return 0;
}
```

在 Xcode 中，你可以通过使用 Product->Perform Action-> Preprocess 来查看任何一个文件的预处理输出。一定要注意这将会花费一些时间来加载预处理输出文件（接近 100，000 行）。

编译

下一个步骤：文本处理和代码生成。我们可以调用 clang 输出汇编代码就像这样：

```
% xcrun clang -S -o - helloworld.c | open -f
```

看一看输出。我们首先注意到的是一些以点开头的行。这些是汇编指令。其他的是真正的 x86_64 汇编代码。最后是些标记，就像 C 中的那些标记一样。

我们从前三行开始：

```
.section    __TEXT,__text,regular,pure_instructions
.globl  _main
.align  4, 0x90
```

这三行是汇编指令，不是汇编代码。”.section” 指令指出了哪一个段接下来将会被执行。比用二进制表示好看多了。

下一个，.global 指令说明_main 是一个外部符号。这就是我们的 main（）函数。它能够从我们的二进制文件之外看到，因为系统要调用它来运行可执行文件。

.align 指令指出了下面代码的对齐方式。从我们的角度看，接下来的代码将会按照 16 比特对齐并且如果需要的时候用 0×90 补齐。

下面是 main 函数的头部：

```
1.  _main:                                  ## @main
2.      .cfi_startproc
3.  ## BB#0:
4.      pushq   %rbp
5.  Ltmp2:
6.      .cfi_def_cfa_offset 16
7.  Ltmp3:
8.      .cfi_offset %rbp, -16
9.      movq    %rsp, %rbp
10. Ltmp4:
11.     .cfi_def_cfa_register %rbp
12.     subq    $32, %rsp
```

这一部分有一些和 C 标记工作机制一样的一些标记。它们是某些特定部分的汇编代码的符号链接。首先是_main 函数真正的开始地址。这个也是被抛出的符号。二进制文件将会在这个地方产生一个引用。

.cfi_startproc 指令一半会在函数开始的地方使用。CFI 是 Call Frame Information 的缩写。帧松散的与一个函数交互。当你使用调试器，并且单步执行的时候，你实际上是在调用帧中跳转。在 C 代码中，函数有自己的调用帧，除了函数之外的一些结构也会有调用站。.cfi_startproc 指令给了函数一个.en_frame 的入口，这个入口包含了堆栈展开信息（表示异常如何展开调用帧堆栈）。这个指令也会发送一些和具体平台相关的指令给 CFI。文件后面的.cfi_endproc 与.cfi_startproc 相匹配，来表示结束 main 函数。

下一步，这里有另外一个 Label ## BB#0.然后，终于来了第一句汇编代码：pushq %rbp。从这里开始事情开始变得有趣。在 OS X 上，我们将会有 x84_64 的代码。对于这种架构，有一个东西叫做 ABI（application binary interface），ABI 表示函数调用是怎样在汇编代码层面上工作的。ABI 指出在函数调用时，rbp 寄存器必须被保护起来。这是 main 函数的责任，来确保返回时，rbp 寄存器中有数据。pushq %rbp 将它的数据推进堆栈，以便我们以后使用。

下面是，两个 CFI 指令： .cfi_def_cfa_offset 16 和 .cfi_offset %rbp, -16. 这将会输出一些信息，这些信息是关于生成调用堆栈展开信息和调试信息的。我们改变了堆栈，并且这两个指令告诉编译器指针指向哪里，或者它们说出了之后调试器将会使用的信息。

现在 movq %rsp， %rbp 将会把局部变量加载进堆栈。subq $32，%rsp 将堆栈指针移动 32 比特，也就是函数将会调用的位置。我们先在 rbp 中存储了老的堆栈指针，然后将此作为我们局部变量的基址，然后我们更新堆栈指针到我们将会使用的位置。

之后，我们调用了 printf（）：

```
1.    leaq        L_.str(%rip),   %rax
2.    movl        $0,   -4(%rbp)
3.    movl        %edi,   -8(%rbp)
4.    movq        %rsi,   -16(%rbp)
5.    movq        %rax,   %rdi
6.    movb        $0,   %al
7.    callq      _printf
```

首先，leaq 加载到 L_.str 的指针到寄存器 rax。注意 L_.str 标记是怎样在下面的代码中定义的。它就是 C 字符串“hello world! \n”。寄存器 edi 和 rsi 保存了函数的第一个和第二个参数。直到我们调用其他函数，我们第一步需要存储它们当前值。这就是为什么我们使用刚刚存储的 rbp 偏移 32 比特的原因。第一个 32 比特是零，之后 32 个比特是 edi 的值（存储了 argc），然后是 64bit 的 rsi 寄存器的值。我们在后面不会使用这些数据。但是如果编译器没有使用优化的时候，它们还是会被存下来。

现在，我们将会把第一个函数（printf）的参数加载进寄存器 edi。printf 函数是一个可变参数的函数。ABI 调用约定指定，将会把使用来存储参数的寄存器数量存储在寄存器 al 中。对我们来讲是 0。最后 callq 调用了 printf 函数。

```
1.    movl        $0,   %ecx
2.    movl        %eax,   -20(%rbp)                   ##  4-byte  Spi
   ll
3.    movl        %ecx,   %eax
```

这将设置 ecx 寄存的值为 0，并且把 eax 的值压栈。然后从 ecx 复制 0 到 eax。A

BI 指定 eax 将会存储函数的返回值，我们 man 函数的返回值是 0：

```
1.    addq        $32,   %rsp
2.    popq        %rbp
3.    ret
4.    .cfi_endproc
```

函数执行完成后，将恢复堆栈指针，通过上移 32bit 在 rsp 中的堆栈指针。我们将会出栈我们早先存储的 rbp 的值，然后调用 ret 来返回，ret 将会读取离开堆栈的地址。.cfi_endproc 平衡了.cfi_startproc 指令。

下一步是一个字一个字的输出我们的字符串：“hello world!\n”：

之后.section 指令指出下面将要跳入的段。L_.str 标记允许获取一个字符转的指针。.asciz 指定告诉汇编器输出一个 0 的字符串结尾。

__TEXT __cstring 开始了一个新的段。这个段包含了 C 字符串：

```
1.    .section        __TEXT,__cstring,cstring_literals
2.    t
   r:
         ##  @.str
3.    .asciz      "Hello  World!\n"
```

这两行创建了一个没有结束符的字符创。注意 L_.str 是怎样命名，和来获取字符串的。

最后.subseciton_via_symbols 指令是静态链接编辑器使用的。

更多关于汇编指令的信息可以从苹果的 Apple’ s assemebler reference 获取。AMD64 网站有关于 ABI for x86 的文档。同时也有 Gentle Introduction to x86-64 Assemble。 再一次，Xcode 允许你查看任何文件的汇编代码通过 Product ->Perform Action -> Assemble.

汇编编译器：
汇编编译器，只是简单的将汇编代码转换成机器码。它创建了一个目标文件。这些文件以.o 结尾。如果你使用 Xcode 构建一个 app，你将会在 Derived Data 目录下面的你的工程目录中的 objects-normal 目录下面发现这些文件。

连接器：
我们将会多谈一点关于链接的东西。但是简单的说，连接器确定了目标文件和库之间的链接。这是什么意思？ 重新调用 callq _printf. printf 是在 libc 库中的一个函数。无论怎样，最后的可执行文件需要能知道 printf（）在内存中的什么位置。例如符号_printf 的地址。连接器将会读取所有的目标文件，所有的库和结束任何未定义的符号。然后将它们编码进最后的可执行文件，然后输出

最后的可执行文件：a.out。

段

就像我们上面提到的一样，这里有些东西叫做段。一个可执行文件包含多个段。可执行文件不同的部分将会加载进不同的段。并且每个段将会转化进一个“Segment”中。这对我们随便写的 app 如此，对我们用心写的 app 也一样。

我们来看看在 a.out 中的段。我们可以使用 size：

```
% xcrun size -x -l -m a.out
Segment __PAGEZERO: 0x100000000 (vmaddr 0x0 fileoff 0)
Segment __TEXT: 0x1000 (vmaddr 0x100000000 fileoff 0)
        Section __text: 0x37 (addr 0x100000f30 offset 3888)
        Section __stubs: 0x6 (addr 0x100000f68 offset 3944)
        Section __stub_helper: 0x1a (addr 0x100000f70 offset 3952)
        Section __cstring: 0xe (addr 0x100000f8a offset 3978)
        Section __unwind_info: 0x48 (addr 0x100000f98 offset 3992)
        Section __eh_frame: 0x18 (addr 0x100000fe0 offset 4064)
        total 0xc5
Segment __DATA: 0x1000 (vmaddr 0x100001000 fileoff 4096)
        Section __nl_symbol_ptr: 0x10 (addr 0x100001000 offset 4096)
        Section __la_symbol_ptr: 0x8 (addr 0x100001010 offset 4112)
        total 0x18
Segment __LINKEDIT: 0x1000 (vmaddr 0x100002000 fileoff 8192)
total 0x100003000
```

a.out 文件有四个段。其中一些有 section。

当我们执行一个可执行文件。虚拟内存系统会将 segment 映射到进程的地址空间中。映射完全不同于我们一般的认识，但是如果你对虚拟内存系统不熟悉，可以简单的想象 VM 会将整个文件加载进内存，虽然在实际上这不会发生。VM 使用了一些技巧来避免全部加载。

当虚拟内存系统进行映射时，数据段和可执行段会以不同的参数和权限被映射。

__TEXT 段包含了可执行的代码。它们被以只读和可执行的方式映射。进程被允许执行这些代码，但是不能修改。这些代码也不能改变它们自己，并且这些页从来不会被污染。

__DATA 段以可读写和不可执行的方式映射。它包含了将会被更改的数据。

第一个段是__PAGEZERO。这个有 4GB 大小。这 4GB 并不是文件的真实大小，但是说明了进程的前 4GB 地址空间将会被映射为，不能执行，不能读，不能写。这就是为什么在去写 NULL 指针或者一些低位的指针的时候，你会得到一个 EXC_BAD_ACCESS 错误。这是操作系统在尝试防止你引起系统崩溃。

在每一个段内有一些片段。它们包含了可执行文件的不同的部分。在_TEXT 段，_text 片段包含了编译得到的机器码。_stubs 和_stub_helper 是给动态链接器用的。着允许动态链接的代码延迟链接。_const 是不可变的部分，就像_cstring 包含了可执行文件的字符串一样。

_DATA 段包含了可读写数据。从我们的角度，我们只有_nl_sysmol_ptr 和__la_symble_ptr,它们是延迟链接的指针。延迟链接的指针被用来执行未定义的函数。例如，那些没有包含在可执行文件本身内部的函数。它们将会延迟加载。那些非延迟链接的指针将会在可执行文件被夹在的时候确定。

其他在_DATA 中共同的段是_const。她包含了那些需要重定位的不可变数据。一个例子是 chat* const p = “foo”; p 指针指向的数据不是静态的。_bss 片段包含了没有被初始化的静态变量例如 static int a; ANSI C 标准指出这些静态变量将会被设置为零。但是在运行时可以被改变。_common 片段包含了被动态链接器使用的占位符片段。

苹果的文档 OSX Assembler Reference 有更多关于片段定义的内容。

段内容：
我们能检查每一个片段的内容，使用 otool 像这样：

```
1.  %  xcrun  otool  -s  __TEXT  __text  a.out
2.  a.out:
3.  (__TEXT,__text)  section
4.  0000000100000f30  55  48  89  e5  48  83  ec  20  48  8d  0
    5  4b  00  00  00  c7
5.  0000000100000f40  45  fc  00  00  00  00  89  7d  f8  48  8
    9  75  f0  48  89  c7
6.  0000000100000f50  b0  00  e8  11  00  00  00  b9  00  00  0
    0  00  89  45  ec  89
7.  0000000100000f60  c8  48  83  c4  20  5d  c3
```

这就是我们 app 的代码。从-s __TEXT __text 非常普通，otool 有一个对此的缩写，使用-t.我们甚至可以看反汇编的代码通过在后面加上-v:

```
% xcrun otool -v -t a.out
a.out:
(__TEXT,__text) section
_main:
0000000100000f30        pushq      %rbp
0000000100000f31        movq        %rsp,  %rbp
0000000100000f34        subq        $0x20,  %rsp
0000000100000f38        leaq        0x4b(%rip),  %rax
0000000100000f3f        movl        $0x0,  0xfffffffffffffffc
(%rbp)
0000000100000f46        movl        %edi,  0xfffffffffffffff8
(%rbp)
0000000100000f49        movq        %rsi,  0xfffffffffffffff0
(%rbp)
0000000100000f4d        movq        %rax,  %rdi
0000000100000f50        movb        $0x0,  %al
0000000100000f52        callq      0x100000f68
0000000100000f57        movl        $0x0,  %ecx
0000000100000f5c        movl        %eax,  0xffffffffffffffec
(%rbp)
0000000100000f5f        movl        %ecx,  %eax
0000000100000f61        addq        $0x20,  %rsp
0000000100000f65        popq        %rbp
0000000100000f66        ret
```

这里有些内容反汇编的代码中的一样，你应该感觉很熟悉，这就是我们在前面编译时候的代码。唯一的不同就是，在这里我们没有任何的汇编指令在里面。这是纯粹的二进制执行文件。

同样的方法，我们可以查案一下其他片段：

```
% xcrun otool -v -s __TEXT __cstring a.out
a.out:
Contents of (__TEXT,__cstring) section
0x0000000100000f8a    Hello World!\n
```

或者：

```
% xcrun otool -v -s __TEXT __eh_frame a.out
a.out:
Contents of (__TEXT,__eh_frame) section
```

```
0000000100000fe0        14 00 00 00 00 00 00 00 01 7a 52 00 01 78 10 01
0000000100000ff0        10 0c 07 08 90 01 00 00
```

关于性能的脚注

从侧面来讲，_DATA 和_TEXT 段会影响性能。如果你有一个非常大的二进制文件，你可能回想查看苹果的代码大小优化指南。将数据移到__TEXT 段是个不错的选择，因为这些页从来不会变脏。

任意的片段

你可以以片段的方式向你的二进制文件添加任何的数据，通过-sectcreate 链接参数。这就是你怎样添加 info.plist 到一个独立的二进制文件。Info.plist 的数据需要被放在_TEXT 段的_info_plist 片段。你可以使用连接器的命令-sectcreate segname sectname file 来实现：

```
-Wl,-sectcreate,__TEXT,__info_plist,path/to/Info.plist
```

同样的，-sectalign 也致命了对齐方式。如果你添加一个全新的段，通过-segprot 来制定数据的保护方式。这些都是在连接器中的帮助手册中的。

你能够到达在/usr/include/mach-o/getsect.h 中定义的函数在二进制文件中的那些片段，通过使用 getsectdata()，它将会返回片段数据的指针和大小。

Mch-o

在 OS X 和 iOS 中可执行文件是 Mach-o 格式的：

```
% file a.out
a.out: Mach-O 64-bit executable x86_64
```

对于 GUI 的程序来说也是这样：

```
% file /Applications/Preview.app/Contents/MacOS/Preview
/Applications/Preview.app/Contents/MacOS/Preview: Mach-O 64-bit executable x86_64
```

你可以从这里找到关于 mach-o 文件格式的详细资料。

我们可以使用 otool 来看一看 mach-o 文件的头部。这说明了这个文件是什么，和怎样被加载的。我们将会使用-h 参数来打印头部信息。

```
% otool -v -h a.out                    a.out:
Mach header
      magic  cputype  cpusubtype    caps        filetype ncmds  sizeofcmds            flags
```

```
4.  MH_MAGIC_64    X86_64                ALL  LIB64           EXEC
    UTE        16              1296      NOUNDEFS  DYLDLINK  TWOLEV
    EL  PIE
```

cputype 和 cpusubtype 指明了可执行文件的目标架构。ncmds 和 sizeofcmds 将会加载一些命令，这些命令我们可以通过-l 参数来查看:

```
1.  %  otool  -v  -l  a.out  |  open  -f
2.  a.out:
3.  Load  command  0
4.              cmd  LC_SEGMENT_64
5.        cmdsize  72
6.        segname  __PAGEZERO
7.          vmaddr  0x0000000000000000
8.          vmsize  0x0000000100000000
9.  ...
```

加载命令指明了文件的逻辑结构和文件在虚拟内存中的布局。绝大多数 otool 打印的信息都是从这些加载命令中来的。看一下 Load comand 1 部分，我们看到了 initprot r-x,这指明了我们上面提到的数据保护模式：只读并且可执行。

对于每一个段和每一个段中的片段,加载命令说明了它们会在内存中的位置和它们的保护模式，例如，这是关于__TEXT __text 片段的输出：

```
1.  Section
2.        sectname  __text
3.          segname  __TEXT
4.               addr  0x0000000100000f30
5.               size  0x0000000000000037
6.             offset  3888
7.              align  2^4  (16)
8.             reloff  0
9.             nreloc  0
10.              type  S_REGULAR
11. attributes  PURE_INSTRUCTIONS  SOME_INSTRUCTIONS
12.   reserved1  0
13.   reserved2  0
```

我们的代码将截止在 0x100000f30.它在文件中的偏移量通常是 3888。如果你看一下 a.out 的范汇编输出。你能够在 0x100000f30 处看到我们的代码。

我们同样可以看一下在可执行文件中，动态链接库是怎样使用的:

```
1.  %  otool  -v  -L  a.out
```

```
2.  a.out:
3.          /usr/lib/libSystem.B.dylib  (compatibility  versio
    n  1.0.0,  current  version  169.3.0)
4.          time  stamp  2  Thu  Jan  1  01:00:02  1970
```

这是你能够在二进制文件中的__printf 符号链接将要用到的库。

一个更复杂的例子

让我们来看一个有三个文件的复杂的例子:

```
1.  Foo.h:
2.  #import  <Foundation/Foundation.h>
3.  @interface  Foo  :  NSObject
4.  -  (void)run;
5.  @end
6.  Foo.m:
7.  #import  "Foo.h"
8.  @implementation  Foo
9.  -  (void)run
10. {
11.         NSLog(@"%@",  NSFullUserName());
12. }
13. @end
14. helloworld.m:
15. #import  "Foo.h"
16. int  main(int  argc,  char  *argv[])
17. {
18.         @autoreleasepool  {
19.                 Foo  *foo  =  [[Foo  alloc]  init];
20.                 [foo  run];
21.                 return  0;
22.         }
23. }
```

编译多个文件

非常明显，我们现在有多个文件。所以我们需要对每一个文件调用 clang 来生成目标文件：

```
1.  %  xcrun  clang  -c  Foo.m
2.  %  xcrun  clang  -c  helloworld.m
```

我们从来不编译头文件。头文件的目的是在实现文件中贡献代码，并通过这种方式来呗编译。通过#import 语句 Foo.m 和 helloworld.m 中都被插入了 foo.h 的内容。 我们得到了两个文件：

```
% file helloworld.o Foo.o
helloworld.o: Mach-O 64-bit object x86_64
Foo.o: Mach-O 64-bit object x86_64
```

为了生成可执行文件，我们需要链接这两个目标文件和 Foundation 系统库：

```
xcrun clang helloworld.o Foo.o -Wl,`xcrun --show-sdk-path`/System/Library/Frameworks/Foundation.framework/Foundation
```

现在，我们可以运行我们的程序了。

符号表和链接

我们这个简单的 app 是通过两个目标文件合并到一起得到的。Foo.o 包含了 Foo 类的实现，同事 helloworld.o 包含了调用 Foo 类方法 run 的 main 函数。 进一步，两个文件都使用了 Foundation 库。在 helloworld.o 中 autorelease pool 使用了这个库，以简洁的方式使用了 libobjc.dylib 中的 Objctive-c 运行时。它需要使用运行时的函数来发送消息调用。foo.o 也是一样的。

这些被形象的称之为符号。我们可以把符号看成一些在运行时将会变成指针的东西。虽然实际上并不是这样能够。 每一个函数，全局变量，类等等都是通过符号的方式来使用的。当我们为可执行文件连接一个目标文件，连接器将会按需要决定目标文件和动态库之间的所有符号。 可执行文件和目标文件都有一个符号表来存储这些符号。如果你使用 nm 工具来查看一下 helloworld.o 你会发现：

```
% xcrun nm -nm helloworld.o
                 (undefined) external _OBJC_CLASS_$_Foo
0000000000000000 (__TEXT,__text) external _main
                 (undefined) external _objc_autoreleasePoolPop
                 (undefined) external _objc_autoreleasePoolPush
                 (undefined) external _objc_msgSend
                 (undefined) external _objc_msgSend_fixup
0000000000000088 (__TEXT,__objc_methname) non-external L_OBJC_METH_VAR_NAME_
000000000000008e (__TEXT,__objc_methname) non-external L_OBJC_METH_VAR_NAME_1
0000000000000093 (__TEXT,__objc_methname) non-external L_OBJC_METH_VAR_NAME_2
00000000000000a0 (__DATA,__objc_msgrefs) weak private external l_objc_msgSend_fixup_alloc
```

```
12.  00000000000000e8  (__TEXT,__eh_frame)  non-external  EH_frame
     0
13.  0000000000000100  (__TEXT,__eh_frame)  external  _main.eh
```

这就是文件中所有的符号链接。__OBJC_CLASS_$_Foo 是类 Foo 的符号链接。它还没有被决定成 Foo 类的外部链接。外部表示它对不是私有的。与此相反 non-external 表明符号链接对于特定的文件是私有的。 我们的 helloworld.o 文件引用了 Foo 类，但是并没有实现它。于是符号最后以未确定结尾。

下面，main 函数同样是外部链接，因为它需要能够被外部看到并被调用。无论怎样，main 函数是在 helloworld 中实现的。并且放在了地址 0，和放在__TEXT __text 片段中。然后是四个 objc 运行时的函数。它们同样是未定义的，需要连接器来决定。

我们再来看看 Foo.o 文件:

```
1.  %  xcrun  nm  -nm  Foo.o
2.  0000000000000000  (__TEXT,__text)  non-external  -[Foo  ru
    n]
3.                                     (undefined)  external  _NSF
    ullUserName
4.                                     (undefined)  external  _NSL
    og
5.                                     (undefined)  external  _OBJ
    C_CLASS_$_NSObject
6.                                     (undefined)  external  _OBJ
    C_METACLASS_$_NSObject
7.                                     (undefined)  external  ___C
    FConstantStringClassReference
8.                                     (undefined)  external  __ob
    jc_empty_cache
9.                                     (undefined)  external  __ob
    jc_empty_vtable
10.  000000000000002f  (__TEXT,__cstring)  non-external  l_.str
11.  0000000000000060  (__TEXT,__objc_classname)  non-external  L_
     OBJC_CLASS_NAME_
12.  0000000000000068  (__DATA,__objc_const)  non-external  l_OBJC
     _METACLASS_RO_$_Foo
13.  00000000000000b0  (__DATA,__objc_const)  non-external  l_OBJC
     _$_INSTANCE_METHODS_Foo
14.  00000000000000d0  (__DATA,__objc_const)  non-external  l_OBJC
     _CLASS_RO_$_Foo
15.  0000000000000118  (__DATA,__objc_data)  external  _OBJC_METAC
     LASS_$_Foo
```

```
0000000000000140 (__DATA,__objc_data) external _OBJC_CLASS_$_Foo
0000000000000168 (__TEXT,__objc_methname) non-external L_OBJC_METH_VAR_NAME_
000000000000016c (__TEXT,__objc_methtype) non-external L_OBJC_METH_VAR_TYPE_
00000000000001a8 (__TEXT,__eh_frame) non-external EH_frame0
00000000000001c0 (__TEXT,__eh_frame) non-external -[Foo run].eh
```

末五行指出_OBJC_CLASS_$_Foo 是一个已定义的并且是个外部符号，同时包含 Foo 的实现。 Foo.o 也有未定义的符号。最前面的是它使用过的 NSFullUserName（），NSLog（）和 NSObject。 当我们连接着两个文件还有 Foundation 库的时候，将会确定这些在动态链接库中的符号。临界期记录了输出文件以来特定的动态链接库和它们的位置。这就是 NSFullName()等将会发生的事情。

我们可以看一下最后的执行文件 a.out 的符号表，就能够发现连接器是怎样确定这些符号的：

```
% xcrun nm -nm a.out
                                 (undefined) external _NSFullUserName (from Foundation)
                                 (undefined) external _NSLog (from Foundation)
                                 (undefined) external _OBJC_CLASS_$_NSObject (from CoreFoundation)
                                 (undefined) external _OBJC_METACLASS_$_NSObject (from CoreFoundation)
                                 (undefined) external ___CFConstantStringClassReference (from CoreFoundation)
                                 (undefined) external __objc_empty_cache (from libobjc)
                                 (undefined) external __objc_empty_vtable (from libobjc)
                                 (undefined) external _objc_autoreleasePoolPop (from libobjc)
                                 (undefined) external _objc_autoreleasePoolPush (from libobjc)
                                 (undefined) external _objc_msgSend (from libobjc)
                                 (undefined) external _objc_msgSend_fixup (from libobjc)
```

```
13.                                     (undefined) external dyld
   _stub_binder (from libSystem)
14. 0000000100000000 (__TEXT,__text) [referenced dynamicall
   y] external __mh_execute_header
15. 0000000100000e50 (__TEXT,__text) external _main
16. 0000000100000ed0 (__TEXT,__text) non-external -[Foo ru
   n]
17. 0000000100001128 (__DATA,__objc_data) external _OBJC_METAC
   LASS_$_Foo
18. 0000000100001150 (__DATA,__objc_data) external _OBJC_CLASS
   _$_Foo
```

我们发现 Foundation 和 Objctive-C 运行时的一些符号依然是未确定的。但是符号表中，记录了怎样去确定它们。例如那些它们可以去查找的动态链接库。

可执行文件一样也知道去哪找这些库：

```
1.  % xcrun otool -L a.out
2.  a.out:
3.          /System/Library/Frameworks/Foundation.framework/Versi
   ons/C/Foundation (compatibility version 300.0.0, current v
   ersion 1056.0.0)
4.          /usr/lib/libSystem.B.dylib (compatibility versio
   n 1.0.0, current version 1197.1.1)
5.          /System/Library/Frameworks/CoreFoundation.framework/V
   ersions/A/CoreFoundation (compatibility version 150.0.0, cu
   rrent version 855.11.0)
6.          /usr/lib/libobjc.A.dylib (compatibility version 1.
   0.0, current version 228.0.0)
```

这些未定义的符号将会在运行时被 dyld(1)确定。当我们执行程序的时候，dyld 将会在 Foundation 中确定指向_NSFullUserName 等的实现的指针，等等等等

我们可以再次使用 nm 来查看你这些符号在 Foundation 中的情况,实际上,如下：

```
1.  % xcrun nm -nm `xcrun --show-sdk-path`/System/Library/Fr
   ameworks/Foundation.framework/Foundation | grep NSFullUserNa
   me
2.  0000000000007f3e (__TEXT,__text) external _NSFullUserNam
   e
```

动态链接编辑器

这里有一些环境变量能帮助我们看一下 dyld 到底做了些什么。首先是 DYLD_PRINT_LIBRARIES.如果设置了，dyld 将会输出已经加载的东戴链接库：

```
1.  %  (export  DYLD_PRINT_LIBRARIES=;  ./a.out  )
2.  dyld:  loaded:  /Users/deggert/Desktop/command_line/./a.out
3.  dyld:  loaded:  /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation
4.  dyld:  loaded:  /usr/lib/libSystem.B.dylib
5.  dyld:  loaded:  /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
6.  dyld:  loaded:  /usr/lib/libobjc.A.dylib
7.  dyld:  loaded:  /usr/lib/libauto.dylib
8.  [...]
```

这显示了七十多个在加载 Foundation 的时候加载的动态链接库。这是因为 Foundation 库也依赖于其他很多动态链接库， 你可以运行：

```
1.  %  xcrun  otool  -L  `xcrun  --show-sdk-path`/System/Library/Frameworks/Foundation.framework/Foundation
```

来查看五十多个 Foundation 依赖的库。

dyld 的共享缓存

当你构建一个真正的程序的时候，你将会链接各种各样的库。它们又会依赖其他的一些框架和动态链接库。于是要加载的动态链接库会非常多。同样非独立的符号也非常多。这里就会有成千上万的符号要确定。这个工作将会话费很多时间——几秒钟。 为了优化这个过程，OS X 和 iOS 上动态链接器使用了一个共享缓存，在/var/db/dyld/。对于每一种架构，操作系统有一个单独的文件包含了绝大多数的动态链接库，这些库已经互相连接并且符号都已经确定。当一个 Mach-o 文件被加载的时候，动态链接器会首先检查共享缓存，如果存在相应的库就是用。每一个进程都把这个共享缓存映射到了自己的地址空间中。这个方法戏剧性的优化了 OS X 和 iOS 上程序的加载时间。

原文：http://www.cocoachina.com/applenews/devnews/2013/1121/7392.html

.NET 项目开发—浅谈面向对象的纵横向关系、多态入口，单元测试（项目小结）

阅读目录：

- 3. 多态入口（面向对象继承体系是可被扩展的）
- 4. 多态的受保护方法的单元测试（Protected 成员的单元测试）

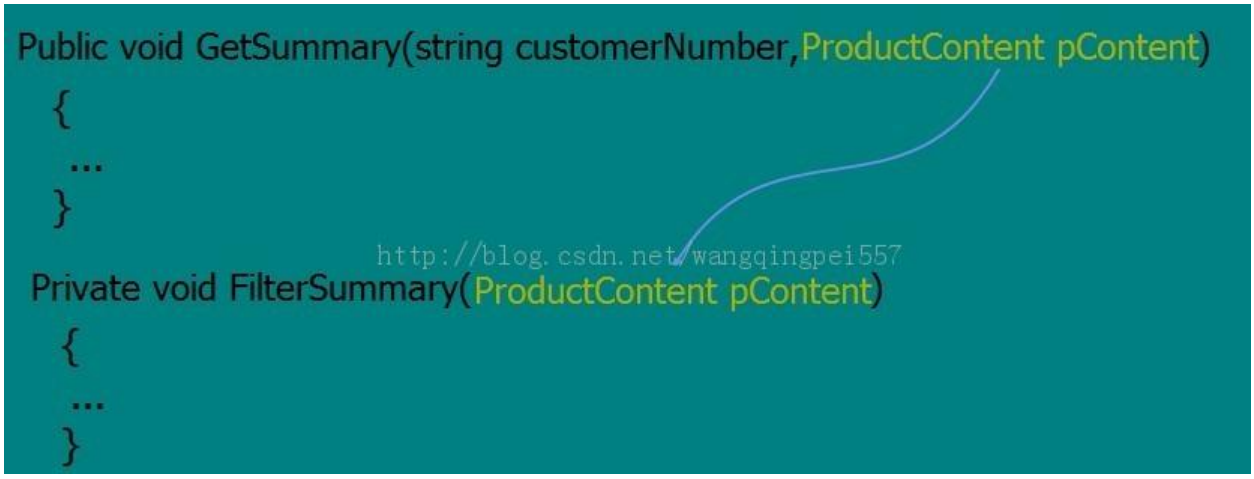
1】开篇介绍

一如既往，这篇文章是我最近在工作中总结出的一点小小的经验，特此写出来与大家分享，因为我觉得日常开发中这些点点滴滴很有用；

2】使用委托消除函数串联调用

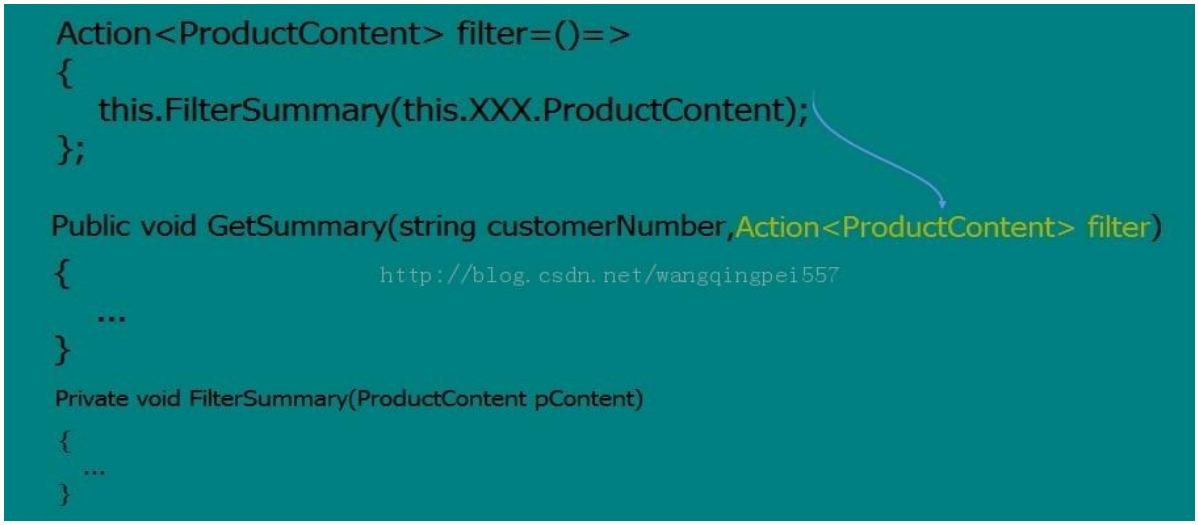
在一般的函数调用情况下，我们都习惯性的将参数传入到某个被调用的方法，这可能就是我们考虑调用方法的惯用思维，但是现在的 C#语言得到了很大的提升，我们可以很自然的使用委托来减少函数之间的参数依赖；有时候会经常看见一个函数的内部逻辑并没有使用到传入的某个参数，而传入的真正目的是为了再传入到本函数需要调用的另外一个函数中去；

图 1：



这个时候我们可以试着使用委托来封装调用的方法，然后将委托实例传入到第一层使用的函数中去，当然要分清使用场景，不是所有的场景都合适；

图 2：



当然需要平衡好这里的内联变量 ProductContent，如果可以的话尽量将委托放入到专门创建委托的委托工厂中去，这样方便全局管理，甚至进一步抽象就可以将委托移除程序硬编码到配置文件；

2.1】使用委托工厂转换两个独立层面的对象

一般情况下，我们在应用层会通过数据访问层的代码获取到数据源中的对应数据实体，然后将其进行 DomainModel 话，只有这样我们才能使用到面向对象的强大功能；这个时候我们只需将创建 DomainModel 的委托工厂构造好，然后作为参数传入到数据访问接口中去；由于应用层是全局协调层，它可以去完成多层之间的协调操作，所以对于应用层的设计可以尽量饱满一点，而不是很简单的一个静态方法集合，这样就会使得 Application Layer 很薄；

3】多态入口（面向对象继承体系是可以被扩展的）

很多时候我们在设计一个框架的时候我们都会注意对象的继承体系，但是我们基本上都没有为这些内部对象留有对外的扩展入口；现假设你有一个框架内部的类 XmlConvert，该类被 XmlConvertSetting 全局静态类引用着，如果不能通过 XmlConvertSetting 对 XmlConvert 进行设置，就无法使用到 XmlConvert 的所有对外提供的扩展方法；

```
1.       public class XmlConvert
2.           {
3.               protected virtual string ConvertReplace(StringBuilder
   NodeString)
4.               {
5.                   return NodeString.ToString().Replace("XXX", "LLL")
   ;
6.               }
7.           }
```

有一个很简单的 XmlConvert 类，是框架内部使用的，现在它提供了一个 Virtual 方法 ConvertReplace，我们想使用这个框架内部的类进行扩展；

```
1.       public class CustomerXmlConvert : XmlConvert
2.       {
3.           protected override string ConvertReplace(StringBuilder Nod
   eString)
4.           {
5.               return base.ConvertReplace(NodeString).Replace("JJJ",
   "AAA");
6.           }
7.       }
```

但是如果未能提供给我们一个多态入口，我们这个自定义的 CustomerXmlConvert 无法起作用；最近发现很多自定义的框架设计上就有这个问题，留有了扩展的类型和相应的方法，但是无法插入到框架内部去，所以特此分享一下；

4】多态的受保护方法的单元测试

受保护方法的单元测试一直都不太好解决，但是我们可以通过简单的继承方式来轻松的处理，就拿上面提到的 XmlConvert 类来举例；

```
1.      public class XmlConvert
2.      {
3.          protected virtual string ConvertReplace(StringBuilder NodeString)
4.          {
5.              return NodeString.ToString().Replace("XXX", "LLL");
6.          }
7.      }
```

如果我们想测试它，直接使用类型继承就可以：

```
1.      [TestClass]
2.      public class XmlConvertTests : XmlConvert
3.      {
4.          [TestMethod]
5.          public void XmlConvert_ConvertReplace_Normal()
6.          {
7.              StringBuilder testData = new StringBuilder("XXXJJJ");

8.              string testResult = this.ConvertReplace(testData);
9.              Assert.AreEqual(testResult, "JJJ");
10.         }
11.     }
```

这里有一个很好的设计启发就是将方法碎片化尽量保持有返回值的操作，这样很好的进行 Assert；其实提到单元测试，冥冥之中总觉得它与面向对象有着一脉相承的感觉，甚至单元测试、重构、面向对象都会起到互补的作用；

内容不多，只是简单的项目小小的总结，希望对大家有用，谢谢；

原文：http://blog.csdn.net/wangqingpei557/article/details/16849483

Volley 使用笔记

Google I/O 2013 上就讲到了 Volley。当时并没还有在意这个类库，直到看了某项目的源代码后，发现这个东西值得推荐。

Volley 这个库的官方介绍是：

```
Volley is a library that makes networking for Android apps
easier and most importantly, faster.
```

不是很严谨的讲，Volley 就是个包含了很多封装功能的网络请求工具类。使用这个工具类有个优势就是可以节省很多在请求以及缓存方面的开发时间。

优势

相比其他网络载入类库，Volley 的优势官方主要提到如下几点：

1. 队列网络请求，并自动合理安排何时去请求。
2. 提供了默认的磁盘和内存等缓存(Disk Caching & Memory Caching)选项。
3. Volley 可以做到高度自定义，它能做到的不仅仅是缓存图片等资源。
4. Volley 相比其他的类库更方便调试和跟踪。

基本使用

引入 Volley 很简单。使用 git 下载代码到本地

```
git clone
https://android.googlesource.com/platform/frameworks/volley
```

然后引入到项目中就可以使用了。

Volley 简单的来讲主要由两个类控制：

1. Request Queue
2. Request

Volley 的「Hello, World」示例代码：

```
// 实例化 Request Queue
RequestQueue queue = Volley.newRequestQueue(context);

// 实例化 Request
String url = "<remote url>";
```

```
JsonObjectRequest jsonObjRequest =
    new JsonObjectRequest(Request.Method.GET, url, null, new
Response.Listener<JSONObject>() {

        @Override
        public void onResponse(JSONObject response) {
            // ...
        }
    }, new Response.ErrorListener() {

        @Override
        public void onErrorResponse(VolleyError error) {
            // ...

        }
    });
```

然后剩下要做的事情就是把这个 Request 扔到 Queue 里面即可：

```
queue.add(jsonObjRequest);
```

缓存图片资源

缓存图片资源 Volley 提供了个自定义的 NetworkImageView 继承自 ImageView 。它的优势就是载入远程图片几乎可以用「傻瓜」形容，例如：

```
mNetworkImageView.setImageUrl(imageUrl, new ImageLoader());
```

其中 ImageLoader 最重要的一个参数就是 ImageLoader.ImageCache 它控制是否需要请求网络获取数据。因此，我们可以将这个 Class 配合 LruCache 以及 DiskLruCache 用来内存和磁盘缓存。

主要方法

```
@Override
public Bitmap getBitmap(String url) {
    Bitmap data = mLruCache.get(url);
    if (data == null) {
        try {
            data = mDiskLruCache.get(key);
            if (data != null) {
                mLruCache.put(key, data);
            }
        } catch (IOException e) {
```

```
                e.printStackTrace();
            }
        }

        return data;
    }
```

这样子，就可以很清晰得把内存缓存和磁盘缓存之间的关系建立和链接起来了。

资源&参考

- http://java.dzone.com/articles/android-%E2%80%93-volley-library
- https://developers.google.com/events/io/sessions/325304728
- https://www.youtube.com/watch?v=yhv8l9F44qo
- https://android.googlesource.com/platform/frameworks/volley

原文：**http://www.gracecode.com/posts/using-volley.html**

HashMap 深度解析(一)

HashMap 可以说是 Java 中最常用的集合类框架之一，是 Java 语言中非常典型的数据结构，我们总会在不经意间用到它，很大程度上方便了我们日常开发。在很多 Java 的笔试题中也会被问到，最常见的，“HashMap 和 HashTable 有什么区别？”，这也不是三言两语能说清楚的，这种笔试题就是考察你来笔试之前有没有复习功课，随便来个快餐式的复习就能给出简单的答案。

HashMap 计划写两篇文章，一篇是 HashMap 工作原理，也就是本文，另一篇是多线程下的 HashMap 会引发的问题。这一年文章写的有点少，工作上很忙，自己业余时间也做点东西，就把博客的时间占用了，以前是力保一周一篇文章，有点给自己任务的意思，搞的自己很累，文章质量也不高，有时候写技术文章也是需要灵感的，为了举一个例子可能要绞尽脑汁，为了一段代码可能要验证好多次，现在想通了，有灵感再写，需要一定的积累，才能把自己了解的知识点总结归纳成文章。

言归正传，了解 HashMap 之前，我们需要知道 Object 类的两个方法 hashCode 和 equals，我们先来看一下这两个方法的默认实现：

[java] view plaincopy

```
1. /** JNI，调用底层其它语言实现 */
2. public native int hashCode();
3.
4. /** 默认同==，直接比较对象 */
5. public boolean equals(Object obj) {
6.     return (this == obj);
7. }
```

equals 方法我们太熟悉了，我们经常用于字符串比较，String 类中重写了 equals 方法，比较的是字符串值，看一下源码实现：

[java] view plaincopy

```
1. public boolean equals(Object anObject) {
2.     if (this == anObject) {
3.         return true;
4.     }
5.     if (anObject instanceof String) {
6.         String anotherString = (String) anObject;
7.         int n = value.length;
8.         if (n == anotherString.value.length) {
9.             char v1[] = value;
10.            char v2[] = anotherString.value;
11.            int i = 0;
12.            // 逐个判断字符是否相等
13.            while (n-- != 0) {
14.                if (v1[i] != v2[i])
15.                        return false;
16.                i++;
17.            }
18.            return true;
19.        }
20.    }
21.    return false;
22.}
```

重写 equals 要满足几个条件：

- **自反性**：对于任何非空引用值 x，x.equals(x) 都应返回 true。
- **对称性**：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。

- **传递性**：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true，并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。
- **一致性**：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false，前提是对象上 equals 比较中所用的信息没有被修改。
- 对于任何非空引用值 x，x.equals(null) 都应返回 false。

Object 类的 equals 方法实现对象上差别可能性最大的相等关系；即，对于任何非空引用值 x 和 y，当且仅当 x 和 y 引用同一个对象时，此方法才返回 true（x == y 具有值 true）。 当此方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相等对象必须具有相等的哈希码。

下面来说说 hashCode 方法，这个方法我们平时通常是用不到的，它是为哈希家族的集合类框架(HashMap、HashSet、HashTable)提供服务，hashCode 的常规协定是：

- 在 Java 应用程序执行期间，在同一对象上多次调用 hashCode 方法时，必须一致地返回相同的整数，前提是对象上 equals 比较中所用的信息没有被修改。从某一应用程序的一次执行到同一应用程序的另一次执行，该整数无需保持一致。
- 如果根据 equals(Object) 方法，两个对象是相等的，那么在两个对象中的每个对象上调用 hashCode 方法都必须生成相同的整数结果。
- 以下情况不 是必需的：如果根据 equals(java.lang.Object) 方法，两个对象不相等，那么在两个对象中的任一对象上调用 hashCode 方法必定会生成不同的整数结果。但是，程序员应该知道，为不相等的对象生成不同整数结果可以提高哈希表的性能。

当我们看到实现这两个方法有这么多要求时，立刻凌乱了，幸好有 IDE 来帮助我们，Eclipse 中可以通过快捷键 alt+shift+s 调出快捷菜单，选择 Generate hashCode() and equals()，根据业务需求，勾选需要生成的属性，确定之后，这两个方法就生成好了，我们通常需要在 JavaBean 对象中重写这两个方法。

好了，这两个方法介绍完之后，我们回到 HashMap。HashMap 是最常用的集合类框架之一，它实现了 Map 接口，所以存储的元素也是键值对映射的结构，并允许使用 null 值和 null 键，其内元素是无序的，如果要保证有序，可以使用 LinkedHashMap。HashMap 是线程不安全的，下篇文章会讨论。HashMap 的类结构如下：

java.util

类 HashMap<K,V>

java.lang.Object

java.util.AbstractMap<K,V>

java.util.HashMap<K,V>

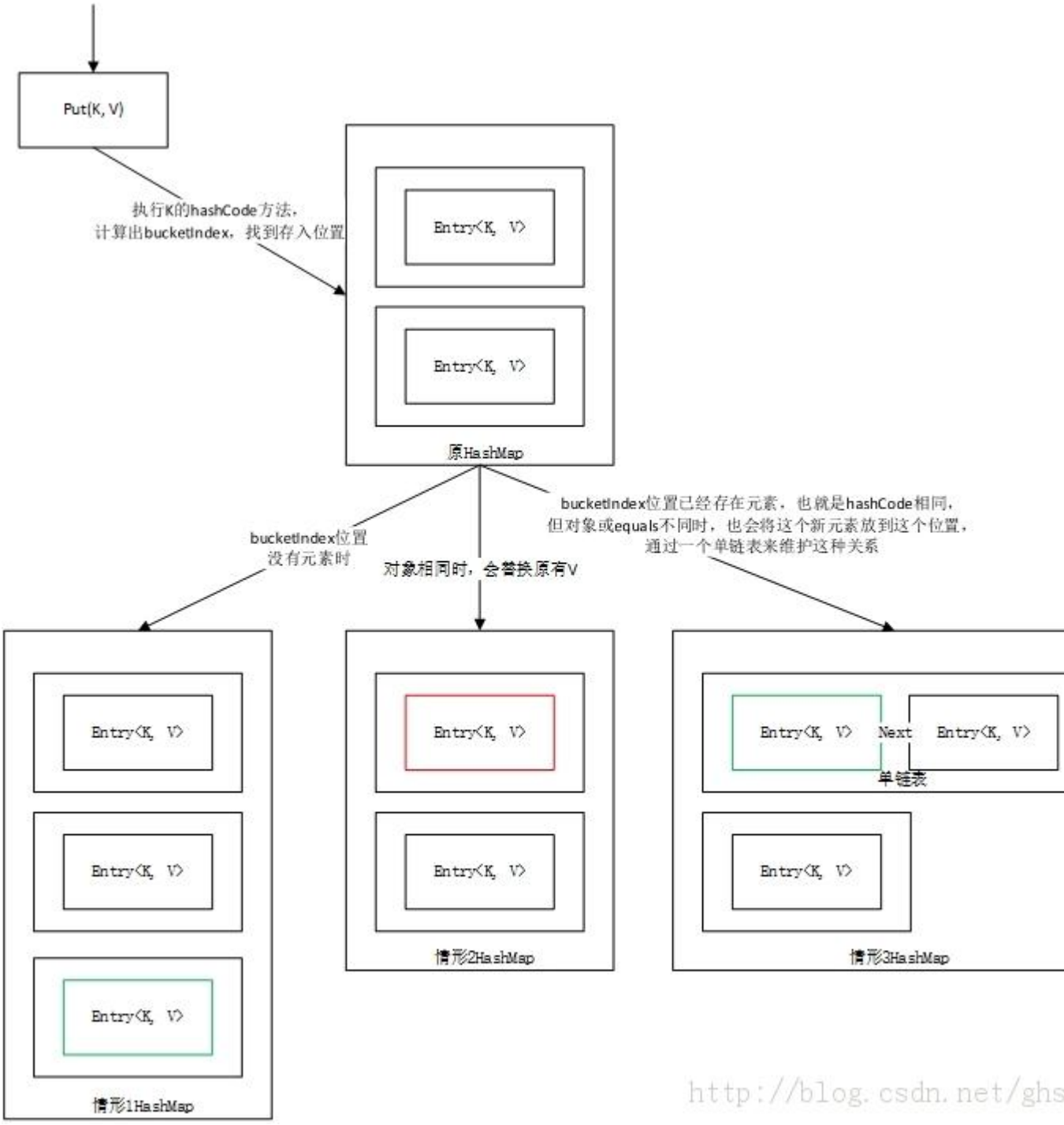
所有已实现的接口：

Serializable,Cloneable,Map<K,V>

直接已知子类：

LinkedHashMap,PrinterStateReasons

HashMap 中我们最长用的就是 put(K, V)和 get(K)。我们都知道，HashMap 的 K 值是唯一的，那如何保证唯一性呢？我们首先想到的是用 equals 比较，没错，这样可以实现，但随着内部元素的增多，put 和 get 的效率将越来越低，这里的时间复杂度是 O(n)，假如有 1000 个元素，put 时需要比较 1000 次。实际上，HashMap 很少会用到 equals 方法，因为其内通过一个哈希表管理所有元素，哈希是通过 hash 单词音译过来的，也可以称为散列表，哈希算法可以快速的存取元素，当我们调用 put 存值时，HashMap 首先会调用 K 的 hashCode 方法，获取哈希码，通过哈希码快速找到某个存放位置，这个位置可以被称之为 **bucketIndex**，通过上面所述 hashCode 的协定可以知道，**如果 hashCode 不同，equals 一定为 false，如果 hashCode 相同，equals 不一定为 true。**所以理论上，hashCode 可能存在冲突的情况，有个专业名词叫**碰撞**，当碰撞发生时，计算出的 **bucketIndex** 也是相同的，这时会取到 **bucketIndex** 位置已存储的元素，最终通过 equals 来比较，equals 方法就是哈希码碰撞时才会执行的方法，所以前面说 HashMap 很少会用到 equals。HashMap 通过 hashCode 和 equals 最终判断出 K 是否已存在，如果已存在，则使用新 V 值替换旧 V 值，并返回旧 V 值，如果不存在 ，则存放新的键值对<K, V>到 **bucketIndex** 位置。文字描述有些乱，通过下面的流程图来梳理一下整个 put 过程。



现在我们知道，执行 put 方法后，最终 HashMap 的存储结构会有这三种情况，情形 3 是最少发生的，哈希码发生碰撞属于小概率事件。到目前为止，我们了解了两件事：

- HashMap 通过键的 hashCode 来快速的存取元素。
- 当不同的对象 hashCode 发生碰撞时，HashMap 通过单链表来解决，将新元素加入链表表头，通过 next 指向旧的元素。单链表在 Java 中的实现就是对象的引用(复合)。

来鉴赏一下 HashMap 中 put 方法源码：

[java] view plaincopy

```
1. public V put(K key, V value) {
2.         // 处理 key 为 null，HashMap 允许 key 和 value 为 null
3.         if (key == null)
4.             return putForNullKey(value);
5.         // 得到 key 的哈希码
6.         int hash = hash(key);
7.         // 通过哈希码计算出 bucketIndex
8.         int i = indexFor(hash, table.length);
9.         // 取出 bucketIndex 位置上的元素，并循环单链表，判断 key 是否已存在
10.        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
11.            Object k;
12.            // 哈希码相同并且对象相同时
13.            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
14.                // 新值替换旧值，并返回旧值
15.                V oldValue = e.value;
16.                e.value = value;
17.                e.recordAccess(this);
18.                return oldValue;
19.            }
20.        }
21.
22.        // key 不存在时，加入新元素
23.        modCount++;
24.        addEntry(hash, key, value, i);
25.        return null;
26.}
```

到这里，我们了解了 HashMap 工作原理的一部分，那还有另一部分，如，加载因子及 rehash，HashMap 通常的使用规则，多线程并发时 HashMap 存在的问题等等，这些会留在下一章说明。

原文：http://blog.csdn.net/ghsau/article/details/16843543

Java 中使用内存映射文件需要考虑的 10 个问题

java 中的 IO 和内存映射文件是什么？

内存映射文件非常特别，它允许 Java 程序直接从内存中读取文件内容，通过将整个或部分文件映射到内存，由操作系统来处理加载请求和写入文件，应用只需要和内存大交道，这使得 IO 操作非常快。加载内存映射文件所使用的内存在 Java 堆区之外。Java 编程语言支持内存映射文件，通过 java.nio 包和 MappedByteBuffer 可以从内存直接读写文件。

内存映射的优缺点

内存映射 IO 最大的优点可能在于性能，这对于建立高频电子交易系统尤其重要。内存映射文件通常比标准通过正常 IO 访问文件要快。另一个巨大的优势是内存映 射 IO 允许加载不能直接访问的潜在巨大文件 。经验表明，内存映射 IO 在大文件处理方面性能更加优异。尽管它也有不足——增加了页面错误的数目。由于操作系统只将一部分文件加载到内存，如果一个请求 页面没有在内存中，它将导致页面错误。同样它可以被用来在两个进程中共享数据。

支持内存映射 IO 的操作系统

大多数主流操作系统比如 Windows 平台，UNIX，Solaris 和其他类 UNIX 操作系统都支持内存映射 IO 和 **64 位架构**，你几乎可以将所有文件映射到内存并通过 JAVA 编程语言直接访问。

Java 的内存映射 IO 的要点

如下为一些你需要了解的 java 内存映射要点:

1. java 通过 java.nio 包来支持内存映射 IO。
2. 内存映射文件主要性能敏感的应用，例如高频电子交易平台。
3. 通过使用内存映射 IO，你可以将大文件加载到内存。
4. 内存映射文件可能导致页面请求错误，如果请求页面不在内存中的话。
5. 映射文件区域的能力取决于于内存寻址的大小。在 32 位机器中，你不能访问超过 4GB 或 2 ^ 32（以上的文件）。
6. 内存映射 IO 比起 Java 中的 IO 流要快的多。
7. 加载文件所使用的内存是 Java 堆区之外，并驻留共享内存，允许两个不同进程共享文件。
8. 内存映射文件读写由操作系统完成，所以即使在将内存写入内存后 java 程序崩溃了，他将仍然会将它写入文件直到操作系统恢复。
9. 出于性能考虑，推荐使用直接字节缓冲而不是非直接缓冲。

10. 不要频繁调用 MappedByteBuffer.force()方法，这个方法意味着强制操作系统将内存中的内容写入磁盘，所以如果你每次写入内存映射文件都调用 force()方法，你将不会体会到使用映射字节缓冲的好处，相反，它(的性能)将类似于同磁盘 IO。
11. 万一发生了电源故障或主机故障，将会有很小的机率发生内存映射文件没有写入到磁盘，这意味着你可能会丢失关键数据。

好吧，小伙伴们，就是这些。内存映射 IO 是高级程序员特别是对于用 Java 编写高性能应用的程序员的一个重要概念。

相关阅读：

Linux 高端内存映射(上) http://www.linuxidc.com/Linux/2012-05/60627.htm
Linux 高端内存映射(中) http://www.linuxidc.com/Linux/2012-05/60628.htm
Linux 高端内存映射(中) http://www.linuxidc.com/Linux/2012-05/60902.htm

原文：http://www.linuxidc.com/Linux/2013-11/92895.htm

【BDTC 讲师】走进“开心农场主”：游戏数据分析的架构及调优

摘要：08 年，他们以《开心农场》打开国际化社交游戏道路，2010 年更携手创新工厂联合创立行云。BDTC 2013 前夕，我们有幸从系统架构、组件调优等方面了解其社交游戏数据分析平台。

编者按：设闹钟、蹲点、外挂，那些年我们为了多偷一点菜可谓是绞尽脑汁。在 BDTC 2013 前夕，通过该公司 CTO 穆黎森和他的工程团队，我们得以从系统架构、组件调优等多个方面了解智明星通的大数据分析平台。

智明星通创立于 2008 年，短短 5 年，已成为中国互联网企业国际化过程中的一家标杆企业。公司现有员工 500+，总部设于北京，并在合肥、香港、台湾、圣保罗（巴西）等地设有分支机构，自己研发的游戏和代理的游戏也在世界各地市场都有不错的表现。

在这些游戏以及其他应用的研发和运营过程中，如何了解应用的运行状况成了一个问题。为了解决这个问题，我们像大多数公司一样，开始组建自己的数据分析团队。到目前为止，行云数据分析平台每日接收和处理的原始日志有 200GB 左右，集群规模在 20 台服务器上下，每天为数十个产品的日常运营提供数据支持。在这个过程中，我们在数据处理技术方面做了很多的探索，在此也希望和大家分享一下。

系统架构

数据处理系统的架构和流程，最基本的框架流程如下图所示：

每天，来自各个项目的数据每时每刻都在打入数据分析系统。这些数据通过数据采集环节，被初步整理成一定的格式，然后录入集中的数据存储系统中。当然，我们系统呈献给用户的是各种统计指标。所谓统计指标，即对原始数据的聚合，形成一定的统计结论。所以，我们需要有查询系统，基于海量的数据进行。在我们的系统中，各个部分的技术选型如下：

紫色的部分即为各个部分的技术实现

客户的应用数据，通过 REST API 打入数据处理系统中。这部分我们考虑使用 PHP 来实现，主要考虑其实现简单，便于水平扩展；处理速度也可以接受。

Rest API 接收到的原始数据，在进入存储之前，还需要进一步整理，主要有两方面的原因：一是一些不合理的日志需要筛选出去；二是我们在这一步，有时候需要把接收到的数据转换成更基本的数据。一个例子是应用程序可能会打入一个应用加载事件 visit。那么，我们收到 visit 的时候，还需要判断这个用户是不是第一次出现；以及用户加载的时候的一些 refer 信息，等等。所以，这一步的逻辑比 Rest API 要复杂,做这些逻辑判断所需要的信息也更多,因此我们选用 java 开发了这一模块。

我们收集到的大量数据，有两方面的信息：一是事件信息，记录发生过的事件：用户 A 在某个时刻登录了一次；用户 B 在另一个时刻进行了一次购买，等等。这种事件信息占了系统所存储信息的大部分，而且绝大部分是不断新增的数据。我们采用 HBase 来存储这部分信息，一方面，HBase 对于批量的写入性能很好；另一方面，在 rowkey 设计合理的情况下，其索引和查询性能也不错。除事件信息外，还有用户的属性信息，比如用户 A 所在地区位于中国，用户 B 首次充值的时间为 2013 年 1 月 1 日，等等。这种信息的访问特点是插入和更新都比较多；在查询的时候，既要按照主键（用户 ID）来查询，也要按照属性值（例如前面提到的充值时间）来查询。这就要求同一个表有多列的索引能力，所以，我们目前采用了 MySQL 来存储用户属性信息。

当数据都导入了以后，我们就需要一种系统，具有从 HBase 和 MySQL 这种异构存储之上进行查询计算的能力。基于原始数据的聚合计算是一个相对来说比较耗资源的过程，在去年，我们主要使用 MapReduce 进行这个计算过程，当计算完成后，把结果保存起来，用于展示。这样做的限制在于，系统所展示的数据只能是预先算好的；用户无法动态地对报告进行修改。基于这个问题，我们很早就开始关注交互式查询技术的进展。今年，分布式环境下的 SQL 查询系统成为一个发展中的热点技术，有若干个这样的系统陆续浮出水面。目前来看，Drill 属于发展较晚的一个项目，但其开放、可扩展的架构能很好的满足我们的需求。所以，我们围绕 Drill，构建了我们的 SQL 查询系统。这样，任何能通过 SQL 表达的数据报告，都可以通过我们的系统来实现了。现在，用户可以通过界面构建任何他想要看到的报告，然后就能在比较短的时间内得到他想要的结果。

上面提到，聚合计算是一个耗资源的过程，即使是现在我们采用了基于 drill 的查询系统缩短了响应时间,计算后的结果也需要保存下来,以应对用户的查询。我们使用 Redis 来保存我们的计算结果，主要考虑 Redis 优秀的随机查询性能。

作为系统的展示层，我们使用了 RoR+backbone+coffeescript 的组合。这个层面选择的自由度就比较大了，coffeescript 应该说是作为 python 爱好者的首选，用来配合 backbone 构建富客户端程序；RoR 用来完成各种界面操作的 Rest API 以支持客户端的运行。总体感觉，这个组合还不错，开发起来也快，同时界面表现也不错。

相关调优

理论上来说，性能调优是一件永无止境的工作。在大数据的背景下，似乎很多问题都可以通过加机器来解决，调优的重要性看起来更低了。但事实上，调优是我们的重要工作之一，原因有以下几点：首先，作为小公司，资源毕竟有限，在集群环境下，调优的成果也具有集群效应；其次，由于我们的系统基于交互式查询技术,那么系统各个环节性能的提升无论对于数据响应的速度还是数据生效的实时性，都会有直接的影响。

所以，我们在系统的各个环节、各个层面都进行了调优。如果用一句废话来解释的话，数据处理系统的特点是（顾名思义）需要处理的数据有很多，所以调优的核心目标只有一个：在尽量短的时间内，使用尽量少的资源（内存，I/O 访问，CPU），处理尽量多的数据。

下面，我们按照模块来做划分，简单介绍一下我们在优化方面所作的努力，囊括数据导入、存储、查询、缓存和前端多个部分。

数据导入部分——主要的优化点在于 java 程序的优化和对于数据库访问的优化（by 刘熊）

关于 Thrift 连接池的优化

在前文中提到，我们在数据导入的时候需要判断一个用户是否为新出现的用户；同时，我们还需要对字符串类型的 ID 做一定的映射，转化为整型的内部表示。我们开发了专门的服务（称之为 ID Service）来处理这种 ID 之间的转换和新 ID 的识别工作。出于性能和开发效率的考虑，我们采用 Apache 的 Thrift RPC 框架来快速搭建 ID Service。不过，Thrift 框架的 Java 客户端本身存在一个不足，就是不支持连接池。为了减少连接创建销毁的开销，我们自己实现了客户端连接池。

关于内存容器类优化

ID Service 的后端存储用的是 MySQL。为了减轻对 MySQL 服务器的压力和提升获取内部整型 ID 的速度，我们采用本地内存来缓存用户原始字符串 ID 和内部整型 ID 的对应关系。用户的原始字符串 ID，哈希成整数后作为关键字来查找对应的内部整型 ID。显然，一个整型到整型的 HashMap 能保存这种对应关系。考虑到 JDK 中 HashMap 的实现相对来说较占内存，我们使用 HPPC 中的 Primitive Collections 来替代。采用这种内存缓存后，经统计，缓存命中率能达到 99%. 如果对 HPPC 中 OpenHashMap 的操作只用到 get/put/remove，那么在每次重新哈希时不用打乱关键字顺序，减少重新哈希的计算时间。

关于 MySQL Connector/J 的优化

每天，我们需要从 MySQL 中查询各个项目两个月以来的活跃用户。对于某些项目，活跃用户数上千万。默认情况下，Connector/J 一次查询会先缓存满足查询条件的所有数据。如果机器内存使用紧张，而某次查询需要缓存的数据量很大，就有可能导致 OOM。Connector/J 支持逐行返回数据（通过 enableStreamingResults 来启用），这样，就能大大减轻短时间内对内存的压力。

我们使用 LOAD DATA LOCAL INFILE 将用户属性信息批量地导入 MySQL。刚开始时，我们把待导入的数据先写到文件中，然后让 LOAD DATA LOCAL INFILE 语句去读这个文件。后来得知，并不需要写入文件，Connector/J 支持从输入流中读取待导入的数据（通过 setLocalInfileInputStream 方法）。这样，批量导入数据就能从内存中读取了，避免了磁盘 IO 开销。经粗略估计，速度能提升 10 倍以上。

关于 HBase 客户端的优化

至于 HBase 批量导入数据，我们在客户端这边做的优化主要是关闭 WAL、调节写缓冲区大小和手动刷新缓冲区。

存储部分——主要的优化点在于对 HBase 和 HDFS 的参数调整，以及访问方式的调整（by 刘熊&王宇飞）

我们在 HBase 参数调优上花了一些时间，主要在如下几个方面作了调整：

- GC：影响 HBase 性能的 GC 参数主要是堆的大小、新生代的大小和垃圾回收器类型。这方面的资料网上较多，前人的经验较多，但不存在万能的最优

参数组合，实践中还是需要根据自己的系统线上的实际表现来不断尝试和优化。

- 数据压缩：HBase 支持启用数据压缩，能节省很多存储空间，但是同时会带来一定的读写性能开销。目前我们线上使用的仍然是 0.94 系列的版本，在 0.96 以及 0.98 版本中，HBase 的存储引擎有非常大的改进，比如引入了新的 prefix tree 编码方式，以及更新的文件结构，等等。这些改动对于性能提升的潜力巨大，我们也十分期待。
- 手动 major compaction：由于执行 major compaction 对 HBase 的响应速度有明显影响，所以我们关闭了自动 major compaction，而改在集群较空闲时执行。
- HDFS 的本地读模式可以让 HBase 的 RegionServer 进程在合适的时候不与本机的 DataNode 进程通讯，而直接从文件系统读取数据。我们打开了本地读模式，提升读性能，好在现在本地读模式已经默认启用了。
- 启用 bloom filter，提升读性能。
- 调节 scan cache 大小，提升 scan 性能。

查询部分——主要的优化点在于如何提高查询的 IO 性能，以及优化查询过程本身（by 王宇飞）

Ad-hoc 查询机制

Direct Scanner

对 hbase 的数据扫描的方式我们做了一些改变，没有使用传统的方式通过 hbase 原生提供的 client 去获取数据，而是设计了一个 direct scanner 的组件来替代 hbase client。

采用 hbase client 的方式会通过 region server 去扫描 hdfs 存放的数据文件（hfile），之后和 region server 本身内存中（memstore）数据做合并，按照 row key 的顺序按批次传给 client。如果数据量小没有明显性能问题，比如扫描一两百万行，但通常我们场景会实时扫描上千万行的数据，这个时候就有明显的性能问题了。主要是 hdfs 上的数据多经过了一层 region server 再传给 client，多增加了一次序列化和网络传输的开销。

Direct scanner 的设计就是为了优化这个问题，我们在 client 的进程中直接扫描 hbase 在 hdfs 上的数据，这个数据占大部分。只从 region server 中获取少部分存在于内存中的数据。最后在 direct scanner 中控制两部分数据的合并和更新覆盖，版本控制等逻辑。

对两种方式的扫描我们做了性能测试对比，发现数据量越大 direct scanner 的优势越明显。当扫描 3000w 行数据时，采用 hbase client 的方式要用到 60s 左右，而 direct scanner 只用 29s。

采样

由于我们本身提供的是一个实时查询的服务，这就要求从查询处理到结果返回给用户的时间要尽可能地短，过长时间的等待对用户的体验会大打折扣。但很多时候用户所需要的统计结果本身涉及到查询的数据量就很大，全部扫描的话光是磁盘 io 的耗费时间用户可能就不能接受。例如，之前提到扫描 3000w 行 hbase 数据 direct scanner 扫描就需要 29s 左右，这里还没有算上物理计划的执行时间。在实际查询中，一个统计指标所涉及到的扫描量可能远不止 3000w 行。

针对这个问题，我们的出发点是想能不能扫描尽量少的数据就能得到相对准确的结果。如果可以，会大大减少不必要的磁盘 io 和 cpu 处理。通过对游戏玩家的行为分析，我们发现大部分事件发生次数和玩家本身有一个比较一致的分布。所以我们通过对玩家的抽样扫描，来预估最终的统计结果。抽样所涉及到的玩家越多，结果也就越准确。所以在实际的查询中，我们采取逐步扫描的方式，每轮都是对一部分玩家采样，直到结果满足误差可以控制在一定范围内为止。

给一个实际的测试结果可能更形象一些，下面是我们扫描某个项目一天发生 visit 事件的人数和事件次数结果，分别用采样和不采样的方式：

不采样： 人数:3187632 次数:82686362 耗时:12.53s

采样（只扫描 1/256 的用户）： 人数:3150336 次数:82599680 耗时：0.49s

可以看到，人数的误差在 1.17%，次数的误差在 0.1%，而查询耗时从 12.52s 减少到了 0.49s。

合并与物理计划执行方式的改进

logical plan 的合并可以说是根据我们的查询业务场景做的一个比较有意思的创新。由于我们前后台交互是异步请求模式，前台提交的查询请求转化为 logical plan 在后台的队列里排队等待处理。而这些排队的 logical plan 如果一个一个的执行可能会涉及到大量对磁盘的重复扫描和计算。

于是我们想有没有方法一次性批量的去执行这些 logical plan，把相同的 io 扫描和计算合并掉。基于这个目的，我们采取的方案是对一系列的 logical plan 做合并，把涉及到重复磁盘扫描的操作符合并到一起。这样多个不同的 logical plan 经过合并后形成的大 logical plan 极大地增强了我们系统的吞吐量。

在有了合并后的 logical plan 后，我们对 drill 原生的物理执行框架又做了进一步改进。Drill 本身的物理执行方式是单线程从 DAG 的 root 节点逐步向 child 节点 pull 数据，这样对于大 plan 的执行就利用不到服务器多核的优势，处理起来就很慢。所以我们把每个 physical operator 节点改造成独立的一个线程 worker，它本身处理完后就把结果 push 给自身的 parent 节点。每个 physical operator 节点有一个数据队列来接收 child 节点所传递过来的数据。通过这种多线程 push 的方式，充分利用了 cpu 的多核能力，加快了大 plan 的处理速度。

缓存部分——主要的优化点是如何考虑缓存的粒度，以提高内存利用率（by 武子竞）

对缓存结果的读取和存放主要为 Get/Set 操作，在低延迟、允许少量数据丢失的情况下，架构上，结果缓存采用 Redis shard 的集群方式，这样可以保证每个 Redis 实例占用较小的系统资源。同时对 Redis 的 save 进行配置，可以灵活掌控数据保存至硬盘的策略。Redis 的硬盘存储文件 dump.rdb 的结构对外开放，也使得扩展集群规模，迁移数据变得相对简便，另外使用 Redis 作为结果缓存，而不用 memcached 的另一个重要原因是 Redis 提供了丰富的数据结构，如 Hash，这对缩减整个结果集的 Keys 的数量也至关重要。

由于每一个查询结果都有对应的 key，因此如果用户定义的查询粒度非常细小，就会导致 key 的数量暴增，因此在缓存的键的设计上，我们选择了折中的粒度，如果是更细粒度的结果，则存储到 Hash 结构中，这样确保了在时间上可接受的前提下，占用内存比 String 更小。使用 hash 的另一个好处是，redis 引入了 zipmap(SmallHash)，可以比正常的 HashMap 节省一些元数据的存储开销，在我们的 hash 中，entry 的数量不大而且相对固定，因此调整 Redis 的 hash-max-zipmap-entries 和 hash-max-zipmap-value 配置可以找到性能和空间的平衡点。

前端——主要是如何使用各种工具来达到更好的交互体验（by 王长历）

采用 web app 方式设计：用 ror 做后台的 rest 服务，页面交互用了 backbone.js 这个前端 MVC 框架，除了报表数据的更新，其他的交互都在浏览器端完成。另外，我们用 coffeescript 来替代 javascript，css 方面采用 bootstrap 框架，部署工具用的是 capistrano。我们也用到了一些很不错的 ruby gem，如利用 resque

用来做异步调用服务，用 private_pub 做消息订阅服务（及时把系统通知推到前端）等等。

相关工具

工具的进步使得小团队的配合变得更加容易。我们的代码均托管在 GitHub（ @XingCloud）上，同时使用 Basecamp 来记录项目中的备忘资源和协调任务。自动化运维方面，开源社区也有非常多的选择，比如我们使用 Ansible 进行配置管理，使用 Fabric 来做集群的自动化升级。

关于未来

数据处理问题三个比较主要的技术环节是：如何整理数据、如何存储数据和如何查询数据。在这三个方面，开源社区目前都处于比较快的进步当中：数据整理方面，各种流式处理框架越来越成熟；存储方面，新的存储格式的探索还在不断进行，老的存储系统比如 HBase 也有很多新的特性在规划当中；查询方面，包括 Drill 的各种查询引擎也在快速发展之中。

针对业务场景我们做了大量的优化工作，原因在于目前还没有一个特别适合且易用的开源数据仓库方案，能够让我们很方便的“开箱即用”。有理由相信，随着开源技术的不断进步，数据处理的门槛将会越来越低，更多的人和组织有能力处理更多的数据；同时新的问题和新的解决方案也会不断出现。作为一名技术人员，能够投身于这一技术潮流中，是一件令人激动的事情。

穆黎森，现智明星通 CTO。曾就职于搜狗，后参加多次创业公司。关注服务器性能优化，关注分布式存储技术，关注分布式查询技术。

Got master's degree in CS from Tsinghua Univ. Served as engineer in sogou.com after graduation. Participated in multiple startup projects, and now entitled as CTO at elex-tech.com, an international game developing

and publishing company. Mainly interested in distributed computation and distributed storage system. （责编/仲浩）

第七届中国大数据技术大会（Big Data Technology Conference 2013，BDTC 2013）将于 2013 年 12 月 5 日-6 日在北京世纪金源大酒店召开。Spark 核心设计者、Databricks 创始人兼 CEO Ion Stoica，Apache HBase 项目管理委员会主席 Michael Stack、百度大数据首席架构师林仕鼎、华为公司诺亚方舟实验室主任杨强、Apache Tez commiter Bikas Saha 大数据技术专家领衔，来自腾讯、阿里巴巴、Hortonworks、LinkedIn、小米、Intel 等 50 余位工程师带来近 60 场干货分享，更有《中国智能交通与大数据技术峰会》专场感受智能交通如何改变生活。

原文：http://www.csdn.net/article/2013-11-21/2817586

跨行清算系统的实现过程

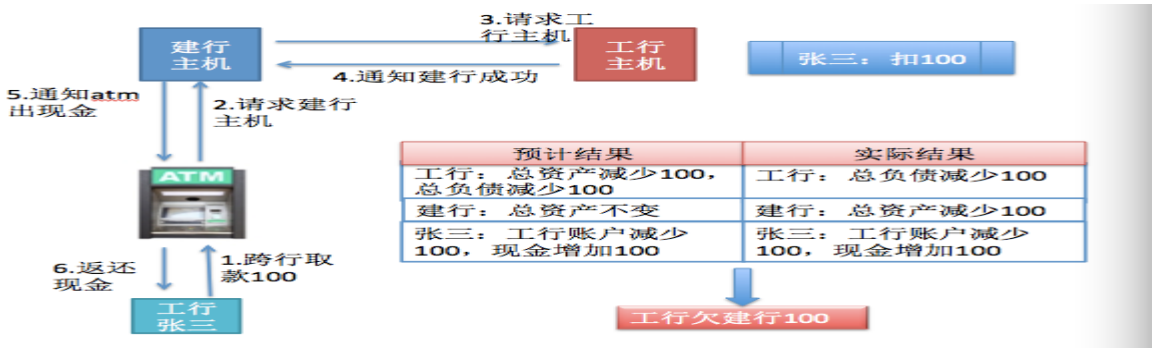
最近看了很多银联方面的清算系统的设计原理，对于跨行清算系统有了很大的了解，写这篇文章的目的是在于从一个程序员的角度去思考一个跨行清算系统的架构是如何实现的以及整个过程中我们有哪些思想是可以借鉴的。由于金融里面涉及到太多的专业名词，包括借贷，备付金，头寸，调拨等等，这里不会涉及到这些，取而代之的是以大家可以理解的概念去解释。

下面简单的介绍一下两种跨行清算系统的实现原理以及特点。一种跨清算系统是我们最熟悉的银联，还有一种是越来越流行的第三方支付系统，比较典型的是快钱。

首先来拿生活中的一个非常常见的例子来说明跨行清算的整个过程，这里面不涉及交易费等其他概念。

跨行取款流程

张三是工行的持卡人，他需要取现金，但是找不到工行的 ATM 机器，发现附近有建行的 ATM 机器，他只能去建行取款，整个过程就是跨行清算的过程，我们以这个场景为例，分析一下业务流程，具体交互流程见下面一张图。



工行持卡人张三在建行 ATM 机器取款 100，ATM 请求建行主机，由于是工行的卡，建行不识别，只能请求工行去处理，工行识别持卡人账户并扣款 100，然后通知建行，建行则通知 atm 吐钱。

这里整个系统要解决两个问题：

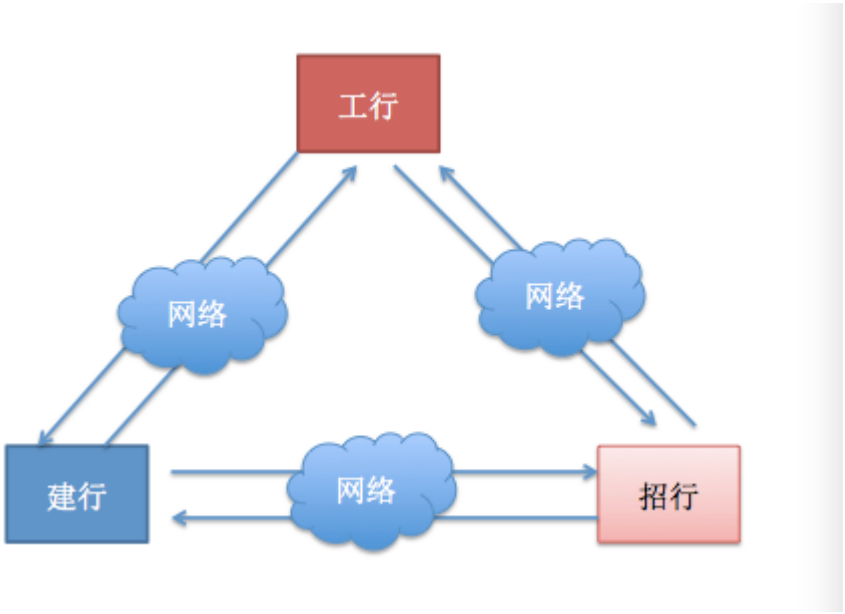
1 建行如何与工行通信

2 建行和工行之间如何清算，如上图结果，工行欠建行 100.

整个系统的分析基于以上两个问题，下面首先解决是通信问题

跨行通信的两种模式

我们先假设工行提供接口，只需要建行发送指约定格式的报文，即可于工行通信，这种相当于建行直接通过接口方式与工行通信。如果是这种方式，只能解决建行和工行的单向通信，如果工行和建行通信，则工行要发送建行指定的通信报文格式。可是大家想想，如果银行更多怎么办，下面是三家银行间的通信



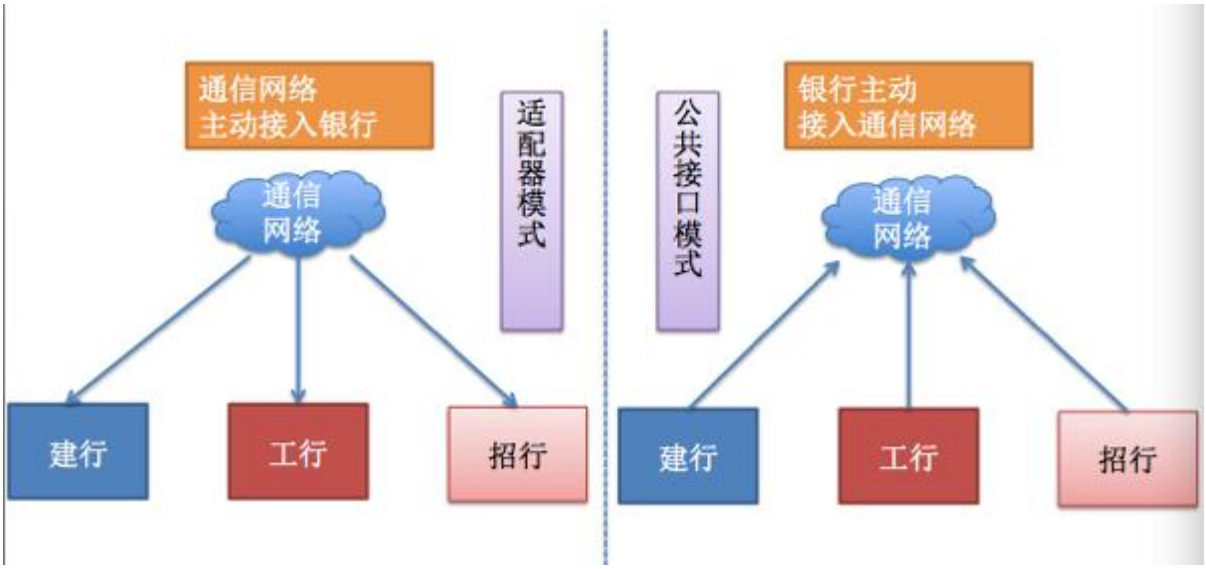
当有三家银行的时候，通信链路就有 3*2=6 条，当银行越来越多的时候，这种点对点的通信变的越来越复杂，每新增一家银行，他要做之前银行都要做的很多重复性的劳动，这样的成本非常高，也不经济，那么必须出现一个网络，它能够接入所有的银行，新的银行只需要接入这个网络，就可以和其他所有的银行进行通信。

先把这个网络成为通信网络，这种通信网络有两种方式可以连接所有的银行

- 1 这个通信网络定义标准接口，所有的银行都必须实现这个通信网络定义的 api，新的银行如果想要接入这个通信网络，必须实现通信接口约定的协议。简称公共接口模式

- 2 这个通信网络主动去连接所有的银行的接口，把所有银行的接口信息都接入里面，就像一个适配器，新的银行如果想要接入这个通信网络，这个通信网络必须主动联系银行，按照银行的接口协议实现通信，简称适配器模式。

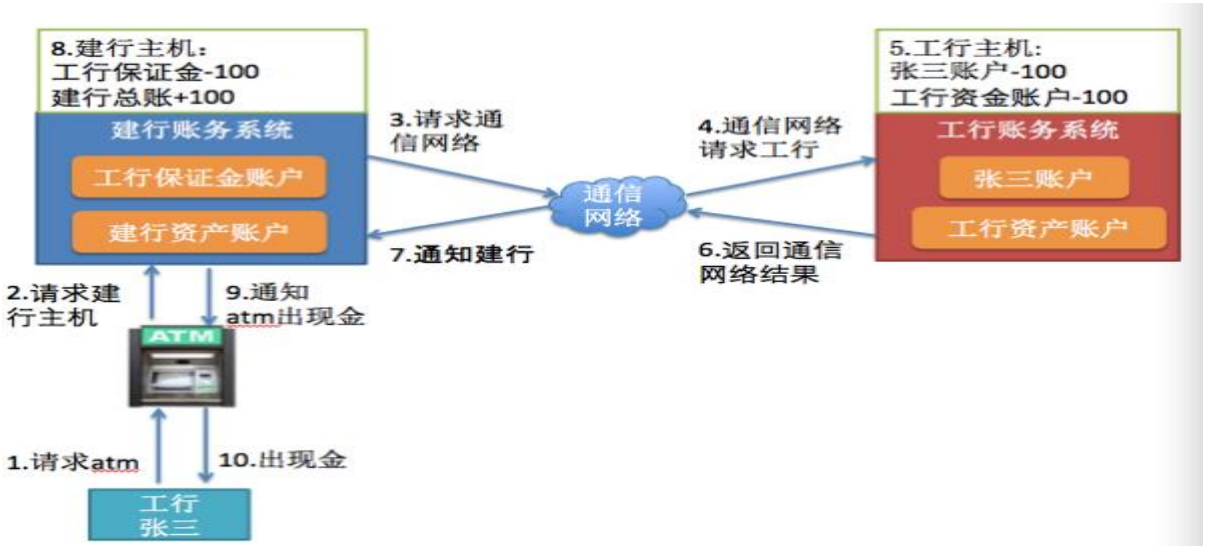
下面一幅图演示了这两种模式的不同：



对于这两模式，主要博弈就在于谁强谁弱。显然第三方支付公司属于适配器模式，需要一家一家银行去接入，至于银联，个人认为应该是第一种模式，这种对于银联这种需要稳定的系统来说是最具有优势的。

跨行清算保证金模式

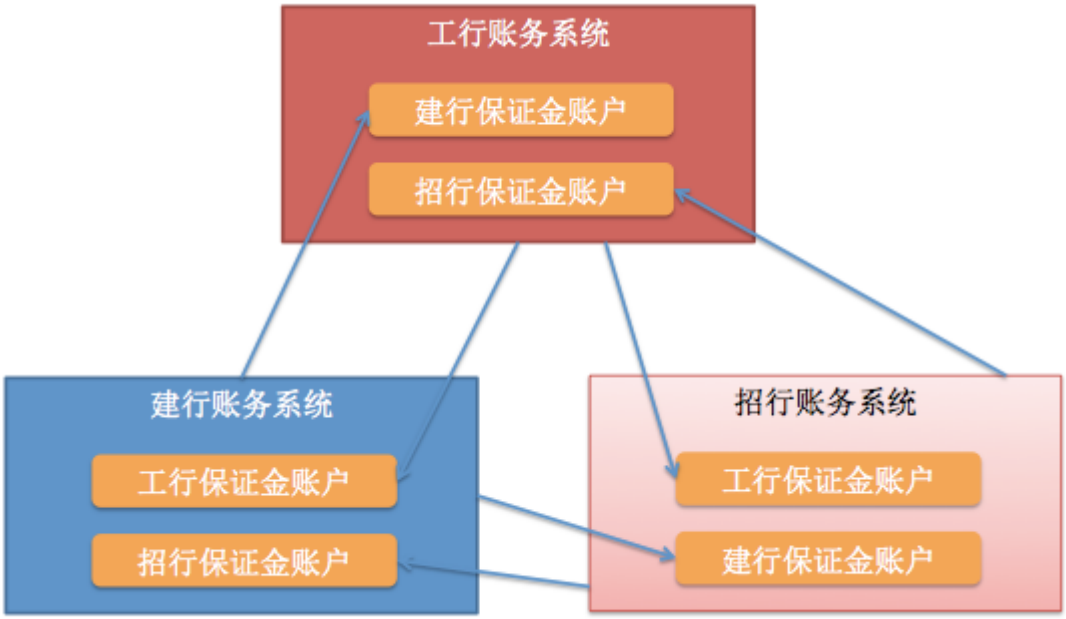
解决了通信问题，下面就看如何解决资金的清算问题。一种简单的方案就是工行在建行里面开设一个保证金账户，用这个账户去偿还在整个跨行交易中应付给建行的资金。



从上图来看，这种方案确实可行。只需要工行在建行里面放足额的保证金，就可以满足跨行的费用。但是这里面实际上存在非常多的问题，

- 1 如果银行越来也多，每个银行都要在其他银行存钱，太不经济了
- 2 保证金需要放多少资金？如果一直都没有发生跨行交易，工行就亏大发了
- 3 如果保证金不够怎么办？交易失败还是记应收款？

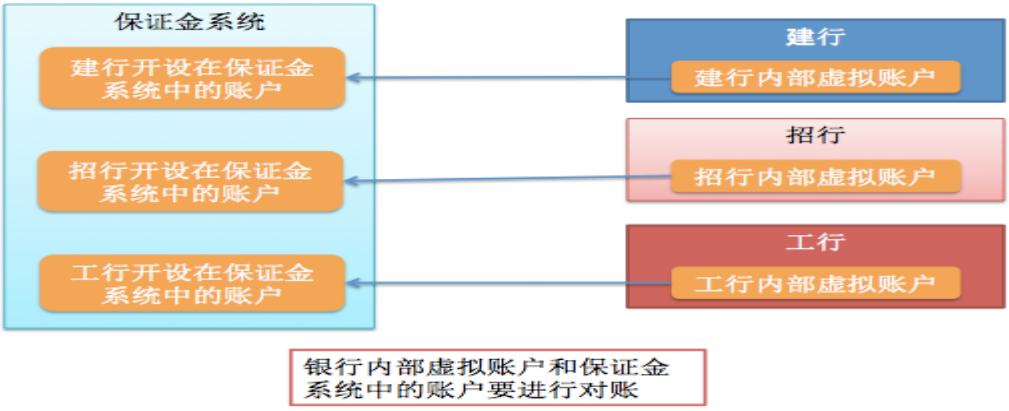
对于第一个问题假设银行越来越多，会导致工行需要在其他每个银行里面都开设保证金账户（见下图），是一个很不经济的方案。



说明这个在其他银行存保证金的方案是不可行的，和之前通信的问题一样，是不是可以把所有的银行保证金账户单独管理起来，统一放置在一起，方便各个银行之间的清算。我们暂时把这个系统称之为保证金系统。

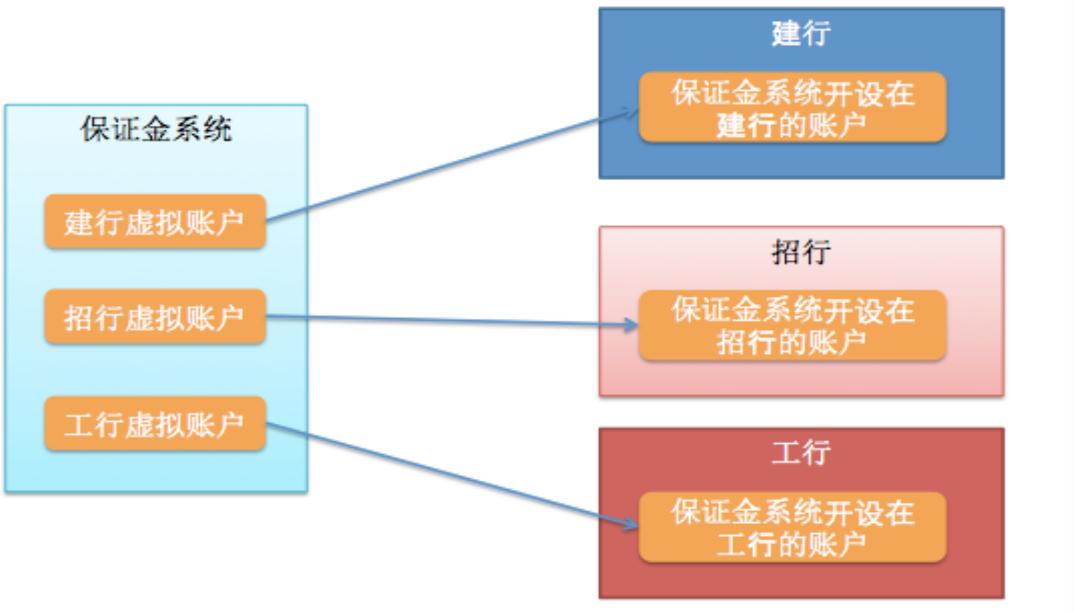
保证金系统

保证金就是方便各个银行之间的清算，需要单独由一个系统进行管理，解决了跨行之间保证金存放的问题。每个银行只需要在保证金系统中存点钱就可以了。保证金系统也有两种模式。先看看比较好理解的第一种模式：



在这种模式下，银行先把一部分钱存放在保证金系统里面，同时银行内部建立一个虚拟账户，记录存放了多少钱，主要是方便对账，万一这个保证金系统钱算错了怎么办。你可以想象一下，银行是很小气的，为啥愿意把钱存放到这保证金系统里面，这部分钱干啥不好，能够银行这么干的只有国家了，这个系统就是央行的备付金管理系统。每个新增的银行都要存一份钱在这里。

另外一种方案是倒过来思考，既然没有牛逼的央行作支撑，那可以在每个商业银行都建立一个账户，用这个账户负责和银行进行清算。每新增一家银行，就在那个银行里面开一个保证金账户。



这两种方式有本质的不同，一个是银行把资金的一部分转出到保证金，银行建立虚拟账户和保证金里面真实的资金映射。一个是保证金系统把资金转出到各个银行，自己内部建立一个虚拟账户和银行中真实的资金账户进行映射。这个间接的银行了后续的对账机制，这里先不叙述。

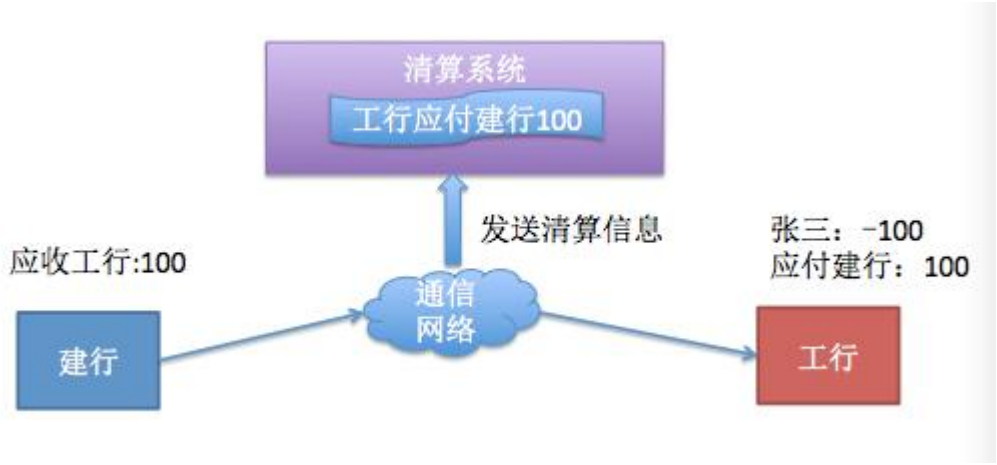
所有的第三方支付公司跨行清算的流程都是第二种方式，只有国家级清算公司（比如银联）是第一种方式，这是一种资源和权力上的不平等，不过是可以理解的。

清算系统

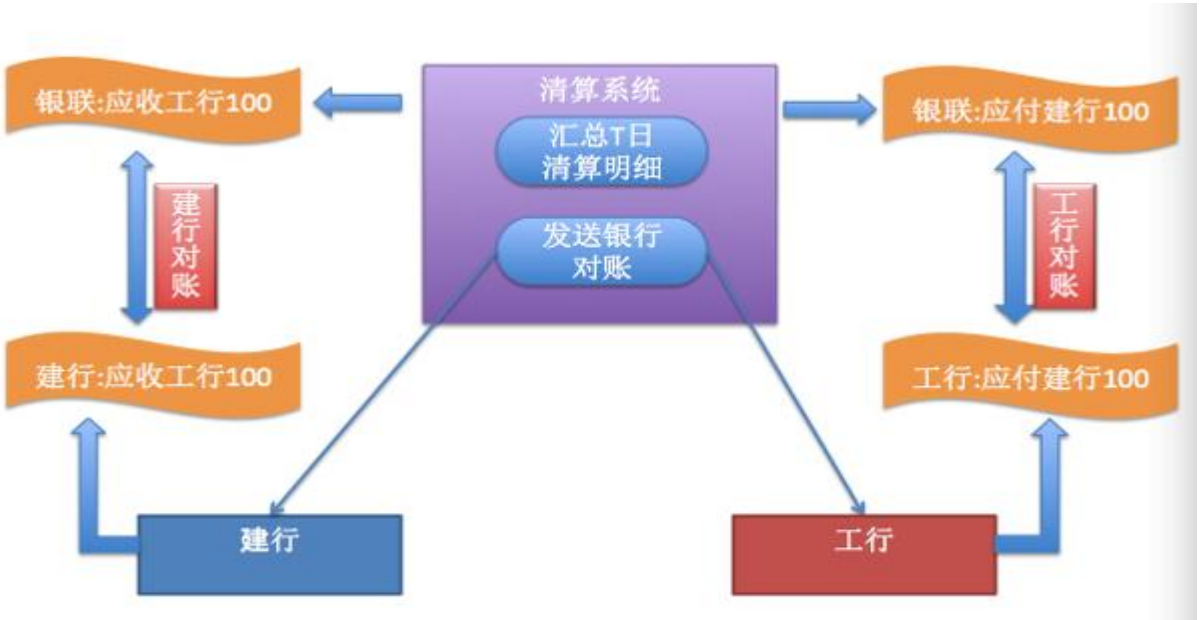
保证金系统解决了保证金存放的问题，接下来就是解决如何清算的问题。假设保证金转账是实时的，就要面对上面说的问题，保证金不够的情况下，跨行交易是成功还是失败。这是一个业务上问题，有很多种解决方案，我们暂不说。从技术上来讲，如果每一笔交易都要保证金实时记账，那么保证金系统的负载太大，事务如何保证等等一些列的问题。所以一个最简单的方案就是：一天结算一次。

每天由一个系统记录这些跨行交易信息，汇总出来，统一记账。这样一天只需要调用一次保证金系统即可。那么整个清算过程则是下面的流程：

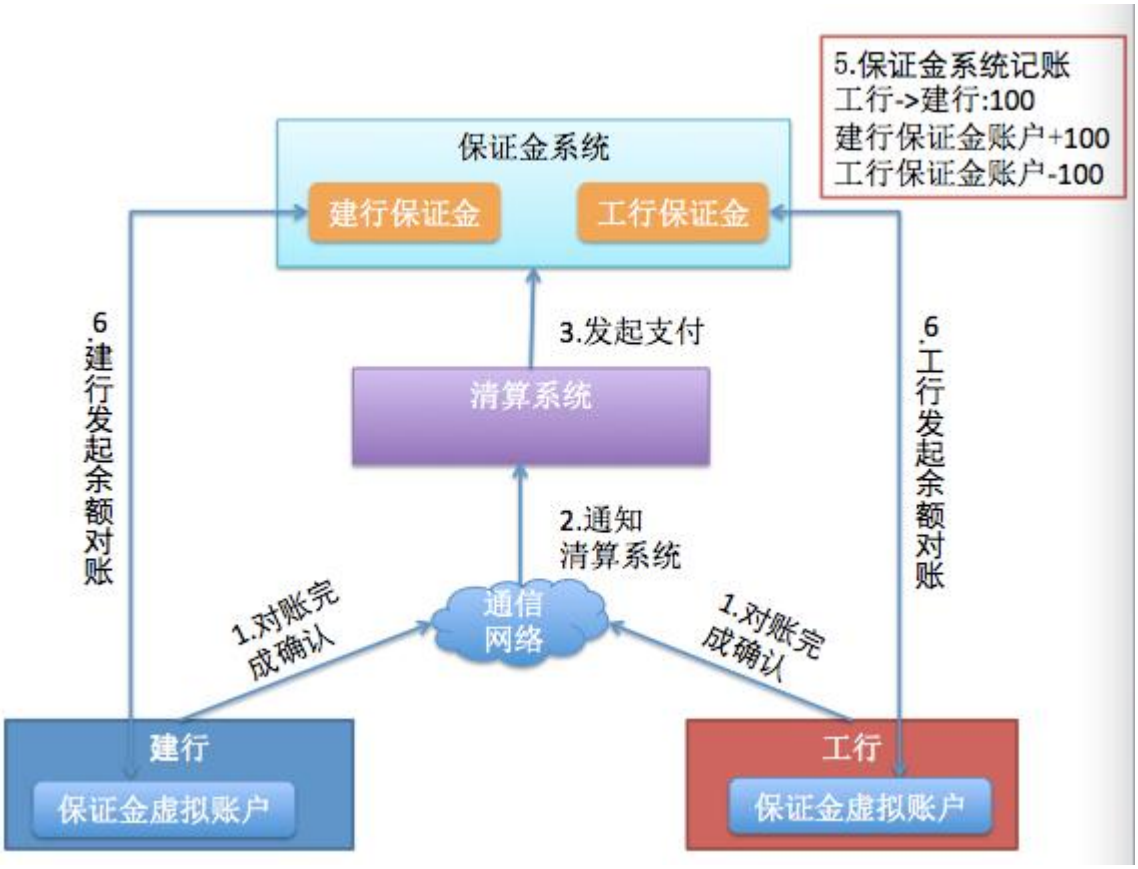
- 1 系统 T 日发生建行和工行的跨行交易 100



- 2 清算系统 T+1 日汇总 T 日工行和建行之间发生的交易明细数据，并且发这些数据发给建行和工行进行确认



- 3 工行建行分别对明细对账确认之后，通知清算系统确认交易明细无误，清算系统开始清算，调用保证金支付系统转账。
- 4 清算完成之后，工行和建行分别获取保证金系统的真实金额和自身系统内部的映射账户进行余额对账。



清算中心最主要干得事情就是统计谁欠谁多少钱，以及触发保证金系统的调拨操作。

对账流程

对账包括两个部分，一个是跨行交易明细的对账以及保证金余额的对账。

首先要思考的是：对账是谁发起的 ？ 这个是了解对账的本质。

我们举生活中的一个例子，我们把钱投资到一个人，那个人负责公司的日常运作。你肯定会主动了解公司的账务，因为那个是你的钱。对账的发起人也是如此，对于银联的清算过程，对账的发起者是商业银行，因为你把钱放在保证金系统里面，这是你的钱，你需要去关心这个的，银联可不关心这个。

对于另外一种保证金系统，把钱放在各个银行里面了，那么对账的发起者就是这个保证金系统维护者了。目前普遍的第三方支付公司都是这个模式，所以他要找各个银行要结果明细进行对账，确认自己的资金安全无误。

以上就是一个简单的跨行清算系统的雏形，从一个就简单的例子入手，说明一个清算过程。目前银联的第三方支付公司的清算过程大致如此，但是实现细节远比这个复杂。但是一个基本的清算系统的本质模型大体上是不会变的。当然这个只是对于同币种的清算，不同币种或者虚拟货币的清算会涉及到汇率的问题，这些就很复杂，有机会在研究一下，后续在分享。

PS：以上很多名词都是自己的随意写的，里面很多专业名词这里不提及，有兴趣的可以自己去了解。

原文：http://www.godiscoder.com/?p=611

Linux 盘符漂移问题

Linux 管理多块磁盘时（以 sata 盘为例），会按磁盘加载的顺序依次给磁盘命名为/dev/sda, /dev/sdb... 这种命名规则就会导致，一块磁盘在发生热插拔或系统重启后，盘符可能发生变化，会影响到一些依赖磁盘盘符工作的应用程序，比如 fstab 里按盘符名来挂载。

要解决磁盘盘符漂移问题，一劳永逸的方法就是将磁盘槽位与盘符名做绑定；淘宝内核组的三百同学针对 ali 内核，添加了磁盘绑定的补丁。

如果只针对磁盘挂载到问题，可通过按标签或 UUID 挂载的方式解决，下文将简单介绍下方案。

如下所示的 fstab，系统启动时，会自动执行每一行挂载动作，将/dev/sda 挂载到/data/disk1，其它依此类推。如果磁盘发生热插拔，第一块磁盘的盘符由原来的/dev/sda 变成了/dev/sdc，那么 fstab 就不能正确挂载第一块磁盘。

```
/dev/sda /data/disk1 ext4 defaults,noatime 0 0
/dev/sdb /data/disk2 ext4 defaults,noatime 0 0
```

为了保证在发生盘符漂移时，磁盘仍能正常挂载，首先对 fstab 做如下改进，按磁盘标签来挂载；比如第一行的含义是，将标签为 disk1 的磁盘挂载到 /data/disk1。

```
LABEL=disk1 /data/disk1    ext4    defaults,noatime  0 0
LABEL=disk2 /data/disk2    ext4    defaults,noatime  0 0
```

接下来的问题就是如何给磁盘设置标签，针对 ext 系列的文件系统，可通过 e*label 来设置标签；也可在磁盘 format 时设置标签。

```
mke4fs /dev/sda -L disk1
或者
mke4fs /dev/sda; e4label /dev/sda disk1
```

通过上述设置后,磁盘/dev/sda 就拥有了标签 disk1,在 fstab 里挂载拥有 disk1 标签的磁盘，即挂载/dev/sda，即使这块磁盘的盘符发生了变化，由于其标签没变，fstab 也能正确的将其挂载；通过 mke4fs 或 e4label 设置的标签，标签实际上是跟文件系统绑定的,是文件系统超级块的一部分,可通过 tune4fs 查询到。

设置标签后，如果磁盘上的文件系统被重新格式化，则其原来设置的标签也就不复存在了,这也正是标签机制不足的地方;如果要解决这个问题,可通过在 fstab 里按 UUID 来挂载磁盘，UUID 对于磁盘来说是不变的，不论其盘符、标签是否变化；但使用 UUID 的缺陷在于灵活性不足，不利于大批量部署。

```
UUID=356fdf58-6923-43d5-9a09-349159c7c8a6 /data/disk1    ext4
defaults,noatime  0 0
UUID=3b93fbad-bea2-4cbb-9a76-b4885924d287 /data/disk1    ext4
defaults,noatime  0 0
```

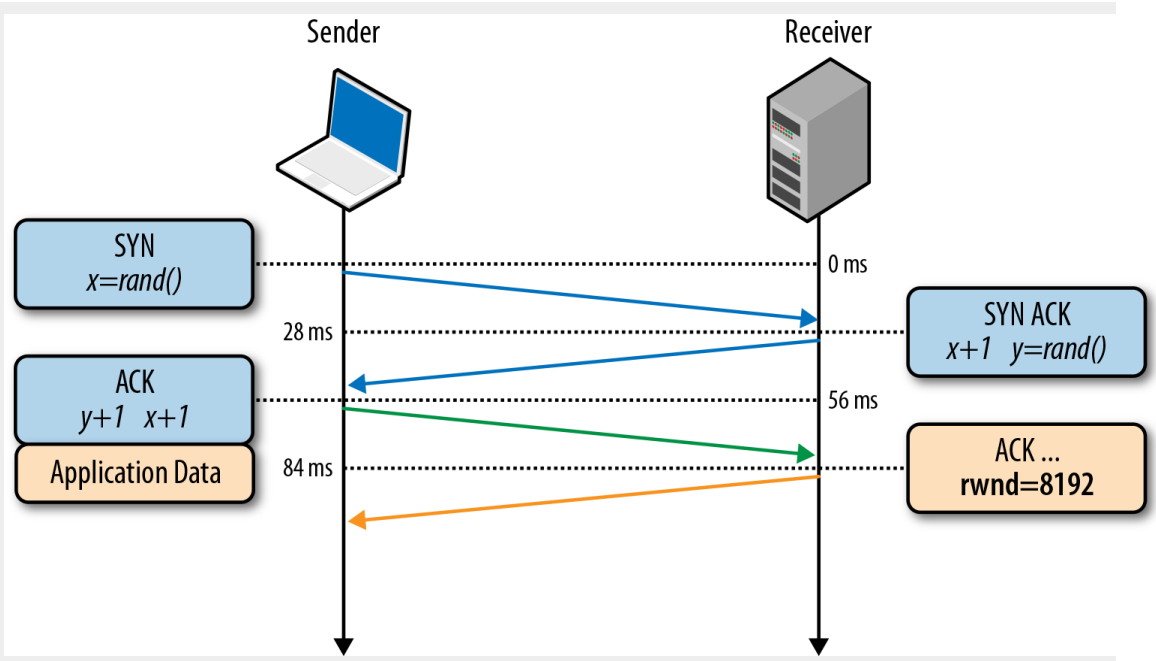
原文：http://blog.yunnotes.net/index.php/linux_disk_name_drift/

浅谈 TCP 优化

很多人常常对 TCP 优化有一种雾里看花的感觉,实际上只要理解了 TCP 的运行方式就能掀开它的神秘面纱。Ilya Grigorik 在「High Performance Browser Networking」中做了很多细致的描述，让人读起来醍醐灌顶，我大概总结了一下，以期更加通俗易懂。

流量控制

传输数据的时候，如果发送方传输的数据量超过了接收方的处理能力，那么接收方会出现丢包。为了避免出现此类问题，流量控制要求数据传输双方在每次交互时声明各自的接收窗口「rwnd」大小，用来表示自己最大能保存多少数据，这主要是针对接收方而言的，通俗点儿说就是让发送方知道接收方能吃几碗饭，如果窗口衰减到零，那么就说明吃饱了，必须消化消化，如果硬撑的话说不定会大小便失禁，那就是丢包了。



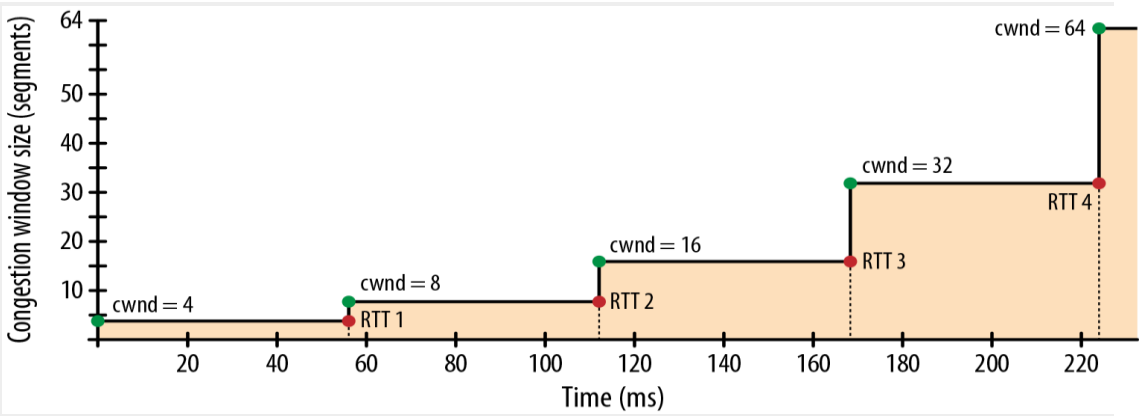
Flow Control

接收方和发送方的称呼是相对的，如果站在用户的角度看：当浏览网页时，数据以下行为主，此时客户端是接收方，服务端是发送方；当上传文件时，数据以上行为主，此时客户端是发送方，服务端是接收方。

慢启动

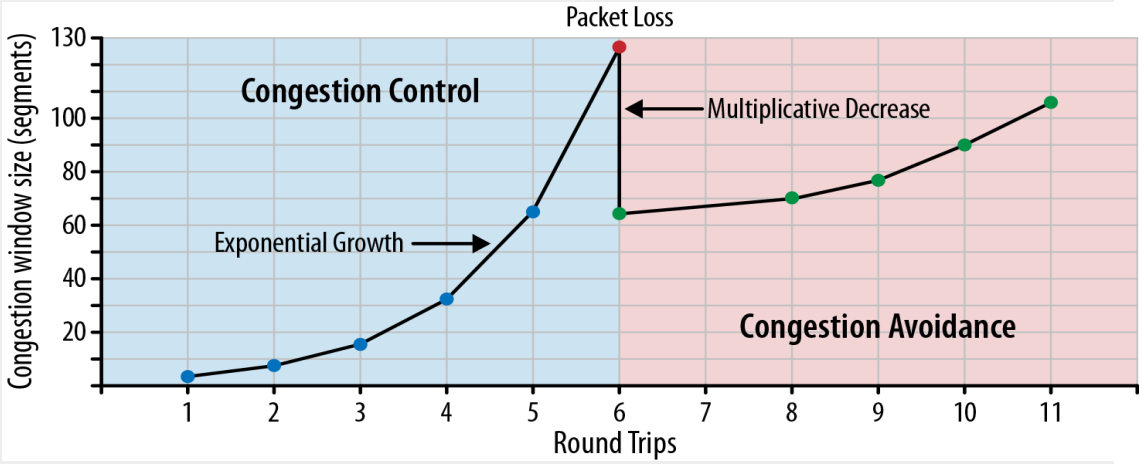
虽然流量控制可以避免发送方过载接收方，但是却无法避免过载网络，这是因为接收窗口「rwnd」只反映了服务器个体的情况，却无法反映网络整体的情况。

为了避免过载网络的问题，慢启动引入了拥塞窗口「cwnd」的概念，用来表示发送方在得到接收方确认前，最大允许传输的未经确认的数据。「cwnd」同「rwnd」相比不同的是：它只是发送方的一个内部参数，无需通知给接收方，其初始值往往比较小，然后随着数据包被接收方确认，窗口成倍扩大，有点类似于拳击比赛，开始时不了解敌情，往往是次拳试探，慢慢心里有底了，开始逐渐加大重拳进攻的力度。



Slow Start

在慢启动的过程中，随着「cwnd」的增加，可能会出现网络过载，其外在表现就是丢包，一旦出现此类问题，「cwnd」的大小会迅速衰减，以便网络能够缓过来。



Congestion Avoidance

说明：网络中实际传输的未经确认的数据大小取决于「rwnd」和「cwnd」中的小值。

拥塞避免

从慢启动的介绍中，我们能看到，发送方通过对「cwnd」大小的控制，能够避免网络过载，在此过程中，丢包与其说是一个网络问题，倒不如说是一种反馈机制，通过它我们可以感知到发生了网络拥塞，进而调整数据传输策略，实际上，这里还有一个慢启动阈值「ssthresh」的概念，如果「cwnd」小于「ssthresh」，那么表示在慢启动阶段；如果「cwnd」大于「ssthresh」，那么表示在拥塞避免阶段，此时「cwnd」不再像慢启动阶段那样呈指数级整整，而是趋向于线性增长，

以期避免网络拥塞，此阶段有多种算法实现，通常保持缺省即可，这里就不一一说明了，有兴趣的读者可以自行查阅。

...

如何调整「rwnd」到一个合理值

有很多人都遇到过网络传输速度过慢的问题，比如说明明是百兆网络，其最大传输数据的理论值怎么着也得有个十兆，但是实际情况却相距甚远，可能只有一兆。此类问题如果剔除奸商因素，多半是由于接收窗口「rwnd」设置不合理造成的。实际上接收窗口「rwnd」的合理值取决于 BDP 的大小，也就是带宽和延迟的乘积。假设带宽是 100Mbps，延迟是 100ms，那么计算过程如下：

```
BDP = 100Mbps * 100ms = (100 / 8) * (100 / 1000) = 1.25MB
```

此问题下如果想最大限度提升吞度量，接收窗口「rwnd」的大小不应小于 1.25MB。说点引申的内容：TCP 使用 16 位来记录窗口大小，也就是说最大值是 64KB，如果超过它，就需要使用 tcp_window_scaling 机制。参考：TCP Windows and Window Scaling。

Linux 中通过配置内核参数里接收缓冲的大小，进而可以控制接收窗口的大小：

```
shell> sysctl -a | grep mem

net.ipv4.tcp_rmem = <MIN> <DEFAULT> <MAX>
```

如果我们出于传输性能的考虑，设置了一个足够大的缓冲，那么当大量请求同时到达时，内存会不会爆掉？通常不会，因为 Linux 本身有一个缓冲大小自动调优的机制，窗口的实际大小会自动在最小值和最大值之间浮动，以期找到性能和资源的平衡点。

通过如下方式可以确认缓冲大小自动调优机制的状态（0：关闭、1：开启）：

```
shell> sysctl -a | grep tcp_moderate_rcvbuf
```

如果缓冲大小自动调优机制是关闭状态，那么就把缓冲的缺省值设置为 BDP；如果缓冲大小自动调优机制是开启状态，那么就把缓冲的最大值设置为 BDP。

实际上这里还有一个细节问题是：缓冲里除了保存着传输的数据本身，还要预留一部分空间用来保存 TCP 连接本身相关的信息，换句话说，并不是所有空间都会被用来保存数据，相应额外开销的具体计算方法如下：

Buffer ／ 2^tcp_adv_win_scale

依照 Linux 内核版本的不同，net.ipv4.tcp_adv_win_scale 的值可能是 1 或者 2，如果为 1 的话，则表示二分之一的缓冲被用来做额外开销，如果为 2 的话，则表示四分之一的缓冲被用来做额外开销。按照这个逻辑，缓冲最终的合理值的具体计算方法如下：

BDP ／ (1 - 1 ／ 2^tcp_adv_win_scale)

此外，提醒一下延迟的测试方法，BDP 中的延迟指的就是 RTT，通常使用 ping 命令很容易就能得到它，但是如果 ICMP 被屏蔽，ping 也就没用了，此时可以试试 synack。

如何调整「cwnd」到一个合理值

一般来说「cwnd」的初始值取决于 MSS 的大小，计算方法如下：
*min(4 * MSS, max(2 * MSS, 4380))*

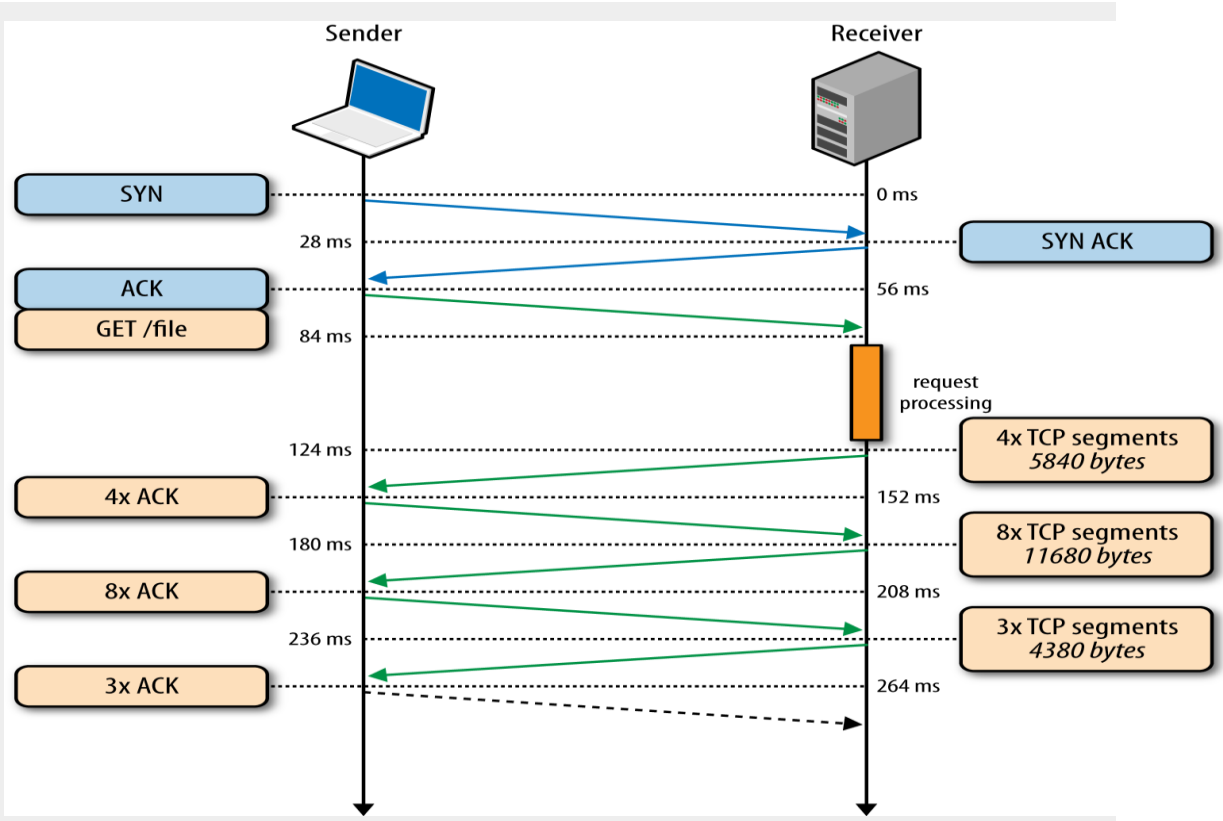
以太网标准的 MSS 大小通常是 1460，所以「cwnd」的初始值是 3MSS。

当我们浏览视频或者下载软件的时候，「cwnd」初始值的影响并不明显，这是因为传输的数据量比较大，时间比较长，相比之下，即便慢启动阶段「cwnd」初始值比较小，也会在相对很短的时间内加速到满窗口，基本上可以忽略不计。

不过当我们浏览网页的时候，情况就不一样了，这是因为传输的数据量比较小，时间比较短，相比之下，如果慢启动阶段「cwnd」初始值比较小，那么很可能还没来得及加速到满窗口，通讯就结束了。这就好比博尔特参加百米比赛，如果起跑慢的话，即便他的加速很快，也可能拿不到好成绩，因为还没等他完全跑起来，终点线已经到了。

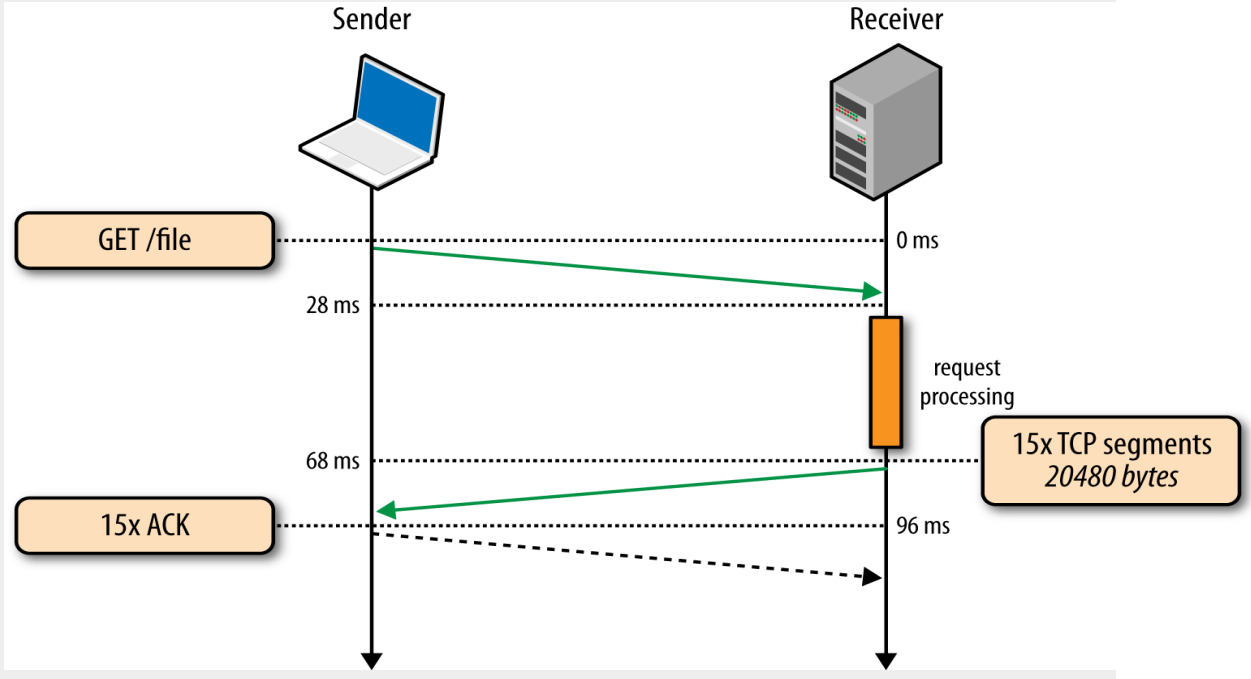
举例：假设网页 20KB，MSS 大小 1460B，如此说来整个网页就是 15MSS。

先让我们看一下「cwnd」初始值比较小（等于 4MSS）的时候会发生什么：



Small Window

再看一下「cwnd」初始值比较大（大于 15MSS）的时候又会如何：



Big Window

明显可见,除去 TCP 握手和服务端处理,原本需要三次 RTT 才能完成的数据传输,当我们加大「cwnd」初始值之后，仅用了一次 RTT 就完成了，效率提升非常大。

推荐：大拿 mnot 写了一个名叫 htracr 的工具，可以用来测试相关的影响。
既然加大「cwnd」初始值这么好，那么到底应该设置多大为好呢？Google 在这方面做了大量的研究，权衡了效率和稳定性之后，最终给出的建议是 10MSS。如果你的 Linux 版本不太旧的话，那么可以通过如下方法来调整「cwnd」初始值：

```
shell> ip route | while read p; do

          ip route change $p initcwnd 10;

       done
```

需要提醒的是片面的提升发送端「cwnd」的大小并不一定有效，这是因为前面我们说过网络中实际传输的未经确认的数据大小取决于「rwnd」和「cwnd」中的小值，所以一旦接收方的「rwnd」比较小的话，会阻碍「cwnd」的发挥。

推荐：相关详细的描述信息请参考：Tuning initcwnd for optimum performance。
有时候我们可能想检查一下目标服务器的「cwnd」初始值设置，此时可以数包：

No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.128.87.3	67.201.31.128	TCP	60590 > http [SYN] Seq
2	0.168582	67.201.31.128	10.128.87.3	TCP	http > 60590 [SYN, ACK
3	0.168618	10.128.87.3	67.201.31.128	TCP	60590 > http [ACK] Seq
4	0.168824	10.128.87.3	67.201.31.128	HTTP	GET /monitor/paessler/
5	0.336917	67.201.31.128	10.128.87.3	TCP	http > 60590 [ACK] Seq
6	0.409468	67.201.31.128	10.128.87.3	TCP	[TCP segment of a reas
7	0.409577	10.128.87.3	67.201.31.128	TCP	60590 > http [ACK] Seq
8	0.409671	67.201.31.128	10.128.87.3	TCP	[TCP segment of a reas
9	0.409677	10.128.87.3	67.201.31.128	TCP	60590 > http [ACK] Seq
10	0.578832	67.201.31.128	10.128.87.3	TCP	[TCP segment of a reas
11	0.578921	10.128.87.3	67.201.31.128	TCP	60590 > http [ACK] Seq
12	0.581443	67.201.31.128	10.128.87.3	TCP	[TCP segment of a reas
13	0.581452	67.201.31.128	10.128.87.3	TCP	[TCP segment of a reas
14	0.581455	67.201.31.128	10.128.87.3	TCP	[TCP segment of a reas
15	0.581490	10.128.87.3	67.201.31.128	TCP	60590 > http [ACK] Seq
16	0.581500	10.128.87.3	67.201.31.128	TCP	60590 > http [ACK] Seq
17	0.581507	10.128.87.3	67.201.31.128	TCP	60590 > http [ACK] Seq
18	0.747147	67.201.31.128	10.128.87.3	TCP	[TCP segment of a reas
19	0.747176	10.128.87.3	67.201.31.128	TCP	60590 > http [ACK] Seq
20	0.749441	67.201.31.128	10.128.87.3	TCP	[TCP segment of a reas
21	0.749467	10.128.87.3	67.201.31.128	TCP	60590 > http [ACK] Seq
22	0.749888	67.201.31.128	10.128.87.3	TCP	[TCP segment of a reas

```
Test Initcwnd
```

通过握手阶段确认 RTT 为 168，开始传输后得到第一个数据包的时间是 409，加上 RTT 后就是 577,从 409 到 577 之间有两个数据包,所以「cwnd」初始值是 2MSS。

需要额外说明的是，单纯数包可能并不准确，因为网卡可能会对包做点手脚，具体说明信息请参考：Segmentation and Checksum Offloading: Turning Off with ethtool。
补充：有人写了一个名叫 initcwnd_check 的脚本，可以帮你检查「cwnd」初始值。
…

实践是检验真理的唯一标准，希望大家多动手，通过实验来检验结果，推荐一篇不错的文章：Impact of Bandwidth Delay Product on TCP Throughput，此外知乎上的讨论也值得一看：为什么多 TCP 连接分块下载比单连接下载快，大家有货的话也请告诉我。

原文：http://huoding.com/2013/11/21/299

淘宝应对双"11"的技术架构分析（专业）

双“11”最热门的话题是 TB，最近正好和阿里的一个朋友聊淘宝的技术架构，发现很多有意思的地方，分享一下他们的解析资料：
数据产品的一个最大特点是数据的非实时写入，正因为如此，我们可以认为，在一定的时间段内，整个系统的数据是只读的。这为我们设计缓存奠定了非常重要的基础。

图 1 淘宝海量数据产品技术架构

按照数据的流向来划分，我们把淘宝数据产品的技术架构分为五层（如图 1 所示），分别是数据源、计算层、存储层、查询层和产品层。位于架构顶端的是我们的数据来源层，这里有淘宝主站的用户、店铺、商品和交易等数据库，还有用户的浏览、搜索等行为日志等。这一系列的数据是数据产品最原始的生命力所在。

在数据源层实时产生的数据，通过淘宝自主研发的数据传输组件 DataX、DbSync 和 Timetunnel 准实时地传输到一个有 1500 个节点的 Hadoop 集群上，这个集群我们称之为“云梯”，是计算层的主要组成部分。在“云梯”上，我们每天有大约 40000 个作业对 1.5PB 的原始数据按照产品需求进行不同的 MapReduce 计算。这一计算过程通常都能在凌晨两点之前完成。相对于前端产品看到的数据，这里的计算结果很可能是一个处于中间状态的结果，这往往是在数据冗余与前端计算之间做了适当平衡的结果。

不得不提的是，一些对实效性要求很高的数据，例如针对搜索词的统计数据，我们希望能尽快推送到数据产品前端。这种需求再采用“云梯”来计算效率将是比较低的，为此我们做了流式数据的实时计算平台，称之为“银河”。“银河”也是一个分布式系统，它接收来自 TimeTunnel 的实时消息，在内存中做实时计算，并把计算结果在尽可能短的时间内刷新到 NoSQL 存储设备中，供前端产品调用。

容易理解，“云梯”或者“银河”并不适合直接向产品提供实时的数据查询服务。这是因为，对于“云梯”来说，它的定位只是做离线计算的，无法支持较高的性能和并发需求；而对于“银河”而言，尽管所有的代码都掌握在我们手中，但要完整地将数据接收、实时计算、存储和查询等功能集成在一个分布式系统中，避免不了分层，最终仍然落到了目前的架构上。

为此，我们针对前端产品设计了专门的存储层。在这一层，我们有基于 MySQL 的分布式关系型数据库集群 MyFOX 和基于 HBase 的 NoSQL 存储集群 Prom，在后面的文字中，我将重点介绍这两个集群的实现原理。除此之外，其他第三方的模块也被我们纳入存储层的范畴。

存储层异构模块的增多，对前端产品的使用带来了挑战。为此，我们设计了通用的数据中间层——glider——来屏蔽这个影响。glider 以 HTTP 协议对外提供 restful 方式的接口。数据产品可以通过一个唯一的 URL 获取到它想要的数据。

以上是淘宝海量数据产品在技术架构方面的一个概括性的介绍，接下来我将重点从四个方面阐述数据魔方设计上的特点。

关系型数据库仍然是王道

关系型数据库（RDBMS）自 20 世纪 70 年代提出以来，在工业生产中得到了广泛的使用。经过三十多年的长足发展，诞生了一批优秀的数据库软件，例如 Oracle、MySQL、DB2、Sybase 和 SQL Server 等。

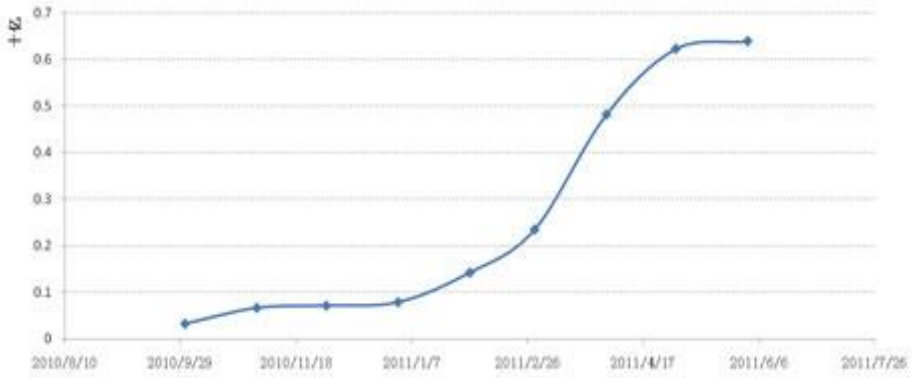


图 2 MyFOX 中的数据增长曲线

尽管相对于非关系型数据库而言，关系型数据库在分区容忍性（Tolerance to Network Partitions）方面存在劣势，但由于它强大的语义表达能力以及数据之间的关系表达能力，在数据产品中仍然占据着不可替代的作用。

淘宝数据产品选择 MySQL 的 MyISAM 引擎作为底层的数据存储引擎。在此基础上，为了应对海量数据，我们设计了分布式 MySQL 集群的查询代理层——MyFOX，使得分区对前端应用透明。

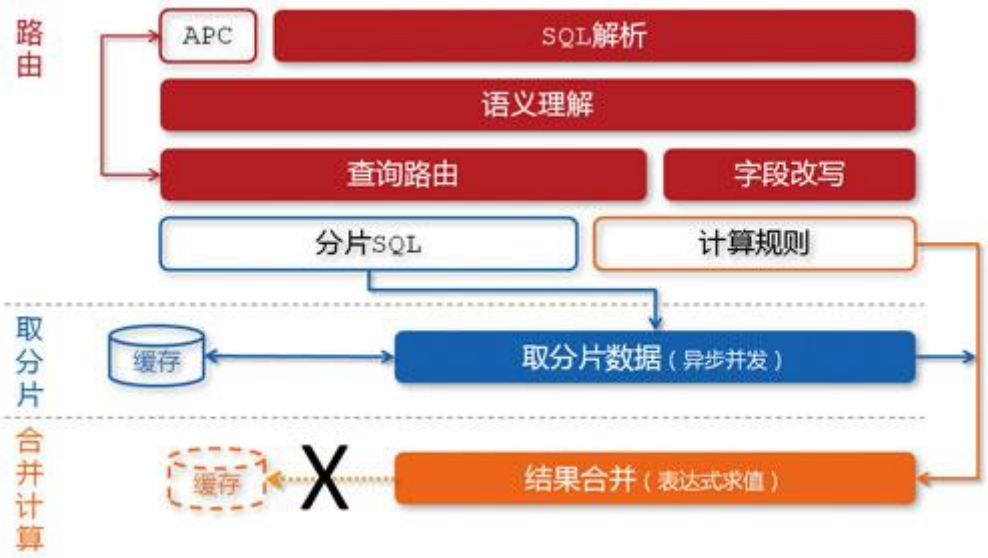


图 3 MyFOX 的数据查询过程

目前，存储在 MyFOX 中的统计结果数据已经达到 10TB，占据着数据魔方总数据量的 95%以上，并且正在以每天超过 6 亿的增量增长着（如图 2 所示）。这些数据被我们近似均匀地分布到 20 个 MySQL 节点上，在查询时，经由 MyFOX 透明地对外服务（如图 3 所示）。

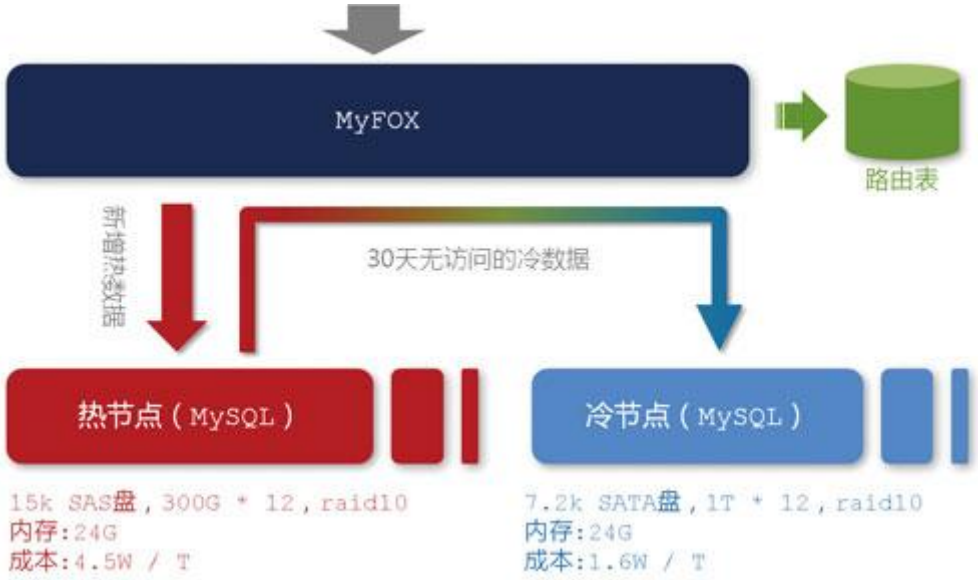


图 4 MyFOX 节点结构

值得一提的是，在 MyFOX 现有的 20 个节点中，并不是所有节点都是“平等”的。一般而言，数据产品的用户更多地只关心“最近几天”的数据，越早的数据，越容易被冷落。为此，出于硬件成本考虑，我们在这 20 个节点中分出了“热节点”和“冷节点”（如图 4 所示）。

顾名思义，“热节点”存放最新的、被访问频率较高的数据。对于这部分数据，我们希望能给用户提供尽可能快的查询速度，所以在硬盘方面，我们选择了每分钟 15000 转的 SAS 硬盘，按照一个节点两台机器来计算，单位数据的存储成本约为 4.5W/TB。相对应地，“冷数据”我们选择了每分钟 7500 转的 SATA 硬盘，单碟上能够存放更多的数据，存储成本约为 1.6W/TB。

将冷热数据进行分离的另外一个好处是可以有效提高内存磁盘比。从图 4 可以看出，“热节点”上单机只有 24GB 内存，而磁盘装满大约有 1.8TB（300＊12＊0.5／1024），内存磁盘比约为 4:300，远远低于 MySQL 服务器的一个合理值。内存磁盘比过低导致的后果是，总有一天，即使所有内存用完也存不下数据的索引了——这个时候，大量的查询请求都需要从磁盘中读取索引，效率大打折扣。

NoSQL 是 SQL 的有益补充

在 MyFOX 出现之后，一切都看起来那么完美，开发人员甚至不会意识到 MyFOX 的存在，一条不用任何特殊修饰的 SQL 语句就可以满足需求。这个状态持续了很长一段时间，直到有一天，我们碰到了传统的关系型数据库无法解决的问题——全属性选择器（如图 5 所示）。

图 5 全属性选择器

这是一个非常典型的例子。为了说明问题，我们仍然以关系型数据库的思路来描述。对于笔记本电脑这个类目，用户某一次查询所选择的过滤条件可能包括“笔记本尺寸”、“笔记本定位”、“硬盘容量”等一系列属性（字段），并且在每个可能用在过滤条件的属性上，属性值的分布是极不均匀的。在图 5 中我们可以看到，笔记本电脑的尺寸这一属性有着 10 个枚举值，而“蓝牙功能”这个属性值是个布尔值，数据的筛选性非常差。

在用户所选择的过滤条件不确定的情况下，解决全属性问题的思路有两个：一个是穷举所有可能的过滤条件组合，在“云梯”上进行预先计算，存入数据库供查询；另一个是存储原始数据，在用户查询时根据过滤条件筛选出相应的记录进行现场计算。很明显，由于过滤条件的排列组合几乎是无法穷举的，第一种方案在现实中是不可取的；而第二种方案中，原始数据存储在什么地方？如果仍然用关系型数据库，那么你打算怎样为这个表建立索引？

这一系列问题把我们引到了“创建定制化的存储、现场计算并提供查询服务的引擎”的思路上来，这就是 Prometheus（如图 6 所示）。

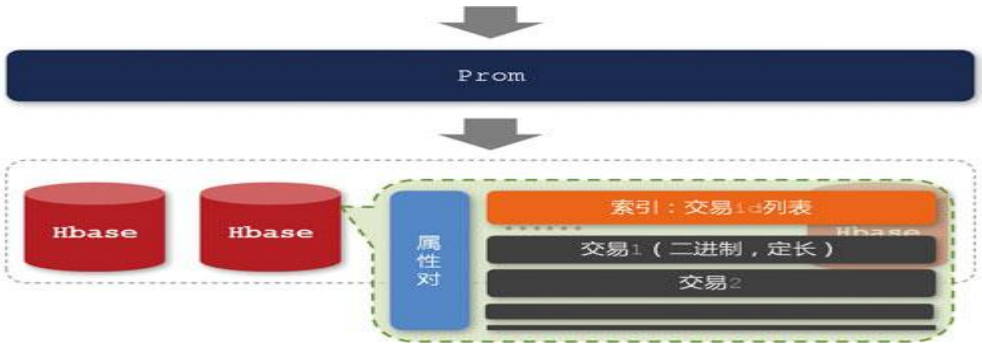


图 6 Prom 的存储结构

从图 6 可以看出，我们选择了 HBase 作为 Prom 的底层存储引擎。之所以选择 HBase，主要是因为它是建立在 HDFS 之上的，并且对于 MapReduce 有良好的编程接口。尽管 Prom 是一个通用的、解决共性问题的服务框架，但在这里，我们仍然以全属性选择为例，来说明 Prom 的工作原理。这里的原始数据是前一天在淘宝上的交易明细，在 HBase 集群中，我们以属性对（属性与属性值的组合）作为 row-key 进行存储。而 row-key 对应的值，我们设计了两个 column-family，即存放交易 ID 列表的 index 字段和原始交易明细的 data 字段。在存储的时候，我们有意识地让每个字段中的每一个元素都是定长的，这是为了支持通过偏移量快速地找到相应记录，避免复杂的查找算法和磁盘的大量随机读取请求。

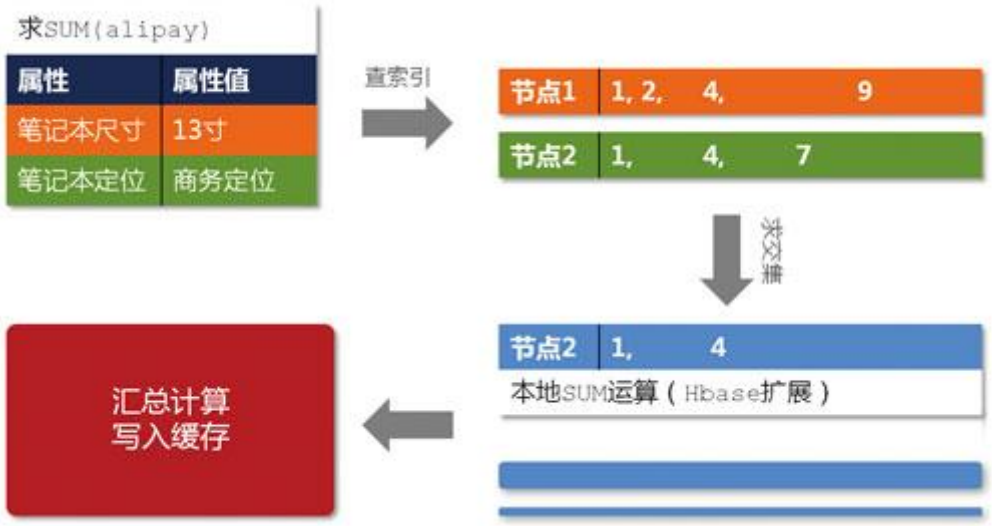


图 7 Prom 查询过程

图 7 用一个典型的例子描述的 Prom 在提供查询服务时的工作原理，限于篇幅，这里不做详细描述。值得一提的是，Prom 支持的计算并不仅限于求和 SUM 运算，统计意义上的常用计算都是支持的。在现场计算方面，我们对 Hbase 进行了扩展，Prom 要求每个节点返回的数据是已经经过“本地计算”的局部最优解，最终的全局最优解只是各个节点返回的局部最优解的一个简单汇总。很显然，这样的设计思路是要充分利用各个节点的并行计算能力，并且避免大量明细数据的网络传输开销。

用中间层隔离前后端

上文提到过，MyFOX 和 Prom 为数据产品的不同需求提供了数据存储和底层查询的解决方案，但随之而来的问题是，各种异构的存储模块给前端产品的使用带来了很大的挑战。并且，前端产品的一个请求所需要的数据往往不可能只从一个模块获取。

举个例子，我们要在数据魔方中看昨天做热销的商品，首先从 MyFOX 中拿到一个热销排行榜的数据，但这里的“商品”只是一个 ID，并没有 ID 所对应的商品描述、图片等数据。这个时候我们要从淘宝主站提供的接口中去获取这些数据，然后一一对应到热销排行榜中，最终呈现给用户。

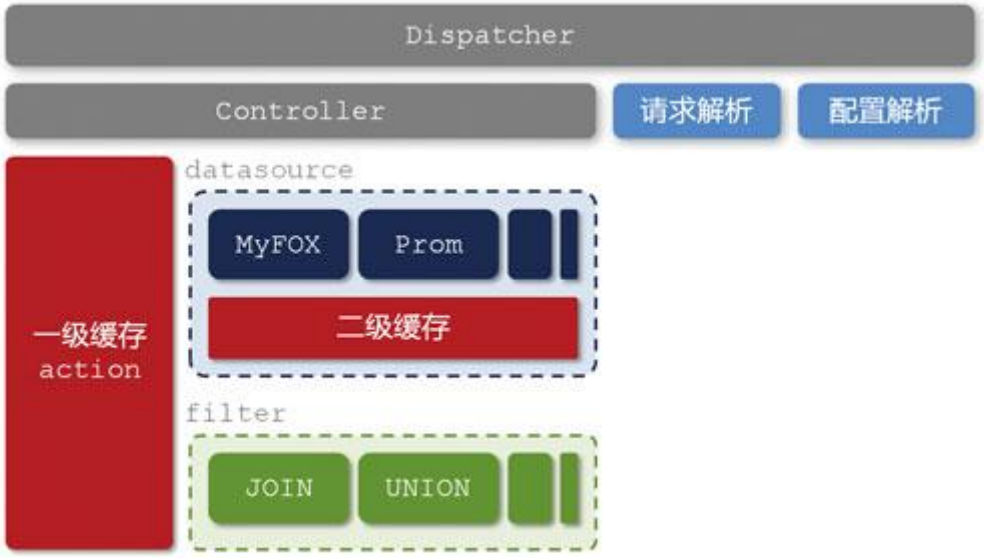


图 8 glider 的技术架构

有经验的读者一定可以想到，从本质上来讲，这就是广义上的异构“表”之间的 JOIN 操作。那么，谁来负责这个事情呢？很容易想到，在存储层与前端产品之间增加一个中间层，它负责各个异构“表”之间的数据 JOIN 和 UNION 等计算，并且隔离前端产品和后端存储，提供统一的数据查询服务。这个中间层就是 glider（如图 8 所示）。

缓存是系统化的工程

除了起到隔离前后端以及异构“表”之间的数据整合的作用之外，glider 的另外一个不容忽视的作用便是缓存管理。上文提到过，在特定的时间段内，我们认为数据产品中的数据是只读的，这是利用缓存来提高性能的理论基础。

在图 8 中我们看到，glider 中存在两层缓存，分别是基于各个异构“表”（datasource）的二级缓存和整合之后基于独立请求的一级缓存。除此之外，各个异构“表”内部可能还存在自己的缓存机制。细心的读者一定注意到了图 3 中 MyFOX 的缓存设计，我们没有选择对汇总计算后的最终结果进行缓存，而是针对每个分片进行缓存，其目的在于提高缓存的命中率，并且降低数据的冗余度。

大量使用缓存的最大问题就是数据一致性问题。如何保证底层数据的变化在尽可能短的时间内体现给最终用户呢？这一定是一个系统化的工程，尤其对于分层较多的系统来说。

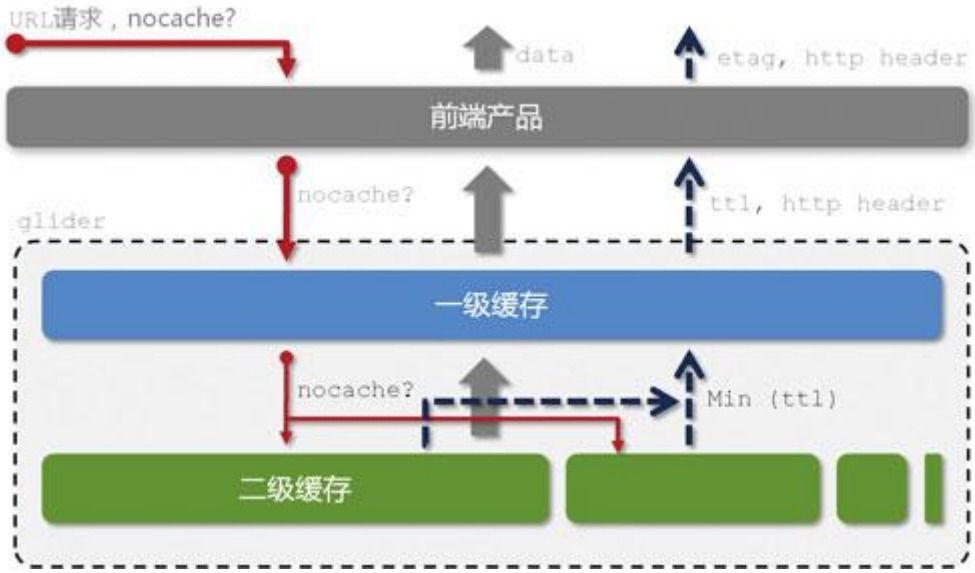


图 9　缓存控制体系

图 9 向我们展示了数据魔方在缓存控制方面的设计思路。用户的请求中一定是带了缓存控制的“命令”的，这包括 URL 中的 query string，和 HTTP 头中的“If-None-Match”信息。并且，这个缓存控制“命令”一定会经过层层传递，最终传递到底层存储的异构“表”模块。各异构“表”除了返回各自的数据之外，还会返回各自的数据缓存过期时间（ttl），而 glider 最终输出的过期时间是各个异构“表”过期时间的最小值。这一过期时间也一定是从底层存储层层传递，最终通过 HTTP 头返回给用户浏览器的。

缓存系统不得不考虑的另一个问题是缓存穿透与失效时的雪崩效应。缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。

有很多种方法可以有效地解决缓存穿透问题，最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。在数据魔方里，我们采用了一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

缓存失效时的雪崩效应对底层系统的冲击非常可怕。遗憾的是，这个问题目前并没有很完美的解决方案。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。在数据魔方中，我们设计的缓存过期机制理论上能够将各个客户端的数据失效时间均匀地分布在时间轴上，一定程度上能够避免缓存同时失效带来的雪崩效应。

结束语

正是基于本文所描述的架构特点，数据魔方目前已经能够提供压缩前 80TB 的数据存储空间，数据中间层 glider 支持每天 4000 万的查询请求，平均响应时间在 28 毫秒（6 月 1 日数据），足以满足未来一段时间内的业务增长需求。

尽管如此，整个系统中仍然存在很多不完善的地方。一个典型的例子莫过于各个分层之间使用短连接模式的 HTTP 协议进行通信。这样的策略直接导致在流量高峰期单机的 TCP 连接数非常高。所以说，一个良好的架构固然能够在很大程度上降低开发和维护的成本，但它自身一定是随着数据量和流量的变化而不断变化的。我相信，过不了几年，淘宝数据产品的技术架构一定会是另外的样子。

其他文章摘要

海量数据领域涵盖分布式数据库、分布式存储、数据实时计算、分布式计算等多个技术方向。

对于海量数据处理，从数据库层面来讲无非就是两点：1、压力如何分摊，分摊的目的就是为了把集中式变为分布式。2、采用多种的存储方案，针对不同的业务数据，不同的数据特点，采用 RDBMS 或采用 KV Store，选择不同数据库软件，使用集中式或分布式存储，或者是其他的一些存储方案。

将数据库进行拆分，包括水平拆分和垂直拆分。

水平拆分主要解决两个问题：1、底层存储的无关性。2、通过线性的去增加机器，支持数据量以及访问请求包括 TPS（Transaction Per Second）、QPS（Query Per Second）的压力增长。其方式如把一张大数据表按一定的方式拆分到不同的数据库服务器上。

海量数据从集中式走向分布式，可能涉及跨多个 IDC 容灾备份特性。

阿里巴巴的数据对不同地域数据的处理方法

由三个产品密切配合解决：是 Erosa、Eromanga 和 Otter。

Erosa 做 MySQL（或其他数据库库）的 Bin-Log 时时解析，解析后放到 Eromanga。Eromanga 是增量数据的发布订阅的产品。Erosa 产生了时时变更的数据发布到 Eromanga。然后各个业务端（搜索引擎、数据仓库或关联的业务方）通过订阅的方式，把时时变更的数据时时的通过 Push 或 Pull 的方式拉到其业务端，进行一些业务处理。而 Otter 就是跨 IDC 的数据同步，把数据能及时反映到不同的 AA 站。

数据同步可能会有冲突，暂时是以那个站点数据为优先，比如说 A 机房的站点的数据是优先的，不管怎么样，它就覆盖到 B 的。

对于缓存。

1、注意切分力度，根据业务选择切分力度。把缓存力度划分的越细，缓存命中率相对会越高。

2、确认缓存的有效生命周期。

拆分策略
1、按字段拆分（最细力度）。如把表的 Company 字段拆掉，就按 COMPANY_ID 来拆。
2、按表来拆，把一张表拆到 MySQL，那张表拆到 MySQL 集群，更类似于垂直拆分。
3、按 Schema 拆分，Schema 拆分跟应用相关的。如把某一模块服务的数据放到某一机群，另一模块服务的数据放到其他 MySQL 机群。但对外提供的整体服务是这些机群的整体组合，用 Cobar 来负责协调处理。

Cobar 类似于 MySQL Proxy，解析 MySQL 所有的协议，相当于可以把它看成 MySQL

Server 来访问的。

原文：http://blog.sina.com.cn/s/blog_4955adb90101nn4b.html

Nginx 战斗准备 —— 优化指南

大多数的 Nginx 安装指南告诉你如下基础知识——通过 apt-get 安装，修改这里或那里的几行配置，好了，你已经有了一个 Web 服务器了！而且，在大多数情况下，一个常规安装的 nginx 对你的网站来说已经能很好地工作了。然而，如果你真的想挤压出 nginx 的性能，你必须更深入一些。在本指南中，我将解释 Nginx 的那些设置可以微调，以优化处理大量客户端时的性能。需要注意一点，这不是一个全面的微调指南。这是一个简单的预览——那些可以通过微调来提高性能设置的概述。你的情况可能不同。

基本的（优化过的）配置

我们将修改的唯一文件是 **nginx.conf**，其中包含 Nginx 不同模块的所有设置。你应该能够在服务器的**/etc/nginx** 目录中找到 nginx.conf。首先，我们将谈论一些全局设置，然后按文件中的模块挨个来，谈一下哪些设置能够让你在大量客户端访问时拥有良好的性能，为什么它们会提高性能。本文的结尾有一个完整的配置文件。

nginx.conf 文件中，Nginx 中有少数的几个高级配置在模块部分之上。

```
user www-data;
pid /var/run/nginx.pid;

worker_processes auto;
```

worker_rlimit_nofile 100000; 高层的配置

user 和 **pid** 应该按默认设置 - 我们不会更改这些内容，因为更改与否没有什么

不同。

worker_processes 定义了 nginx 对外提供 web 服务时的 worder 进程数。最优值取决于许多因素，包括（但不限于）CPU 核的数量、存储数据的硬盘数量及负载模式。不能确定的时候，将其设置为可用的 CPU 内核数将是一个好的开始（设置为“auto”将尝试自动检测它）。

worker_rlimit_nofile 更改 worker 进程的最大打开文件数限制。如果没设置的话，这个值为操作系统的限制。设置后你的操作系统和 Nginx 可以处理比“ulimit -a”更多的文件，所以把这个值设高，这样 nginx 就不会有“too many open files”问题了。

Events 模块

events 模块中包含 nginx 中所有处理连接的设置。

```
events {
    worker_connections 2048;
    multi_accept on;
    use epoll;
}
```

worker_connections 设置可由一个 worker 进程同时打开的最大连接数。如果设置了上面提到的 worker_rlimit_nofile，我们可以将这个值设得很高。

记住，最大客户数也由系统的可用 socket 连接数限制（~ 64K），所以设置不切实际的高没什么好处。

multi_accept 告诉 nginx 收到一个新连接通知后接受尽可能多的连接。

use 设置用于复用客户端线程的轮询方法。如果你使用 Linux 2.6+，你应该使用 epoll。如果你使用*BSD，你应该使用 kqueue。想知道更多有关事件轮询？看下维基百科吧（注意，想了解一切的话可能需要 neckbeard 和操作系统的课程基础）

（值得注意的是如果你不知道 Nginx 该使用哪种轮询方法的话，它会选择一个最适合你操作系统的）

HTTP 模块

HTTP 模块控制着 nginx http 处理的所有核心特性。因为这里只有很少的配置，所以我们只节选配置的一小部分。所有这些设置都应该在 http 模块中，甚至你不会特别的注意到这段设置。

```
http {

    server_tokens off;

    sendfile on;

    tcp_nopush on;
    tcp_nodelay on;

    ...
}
```

server_tokens 并不会让 nginx 执行的速度更快,但它可以关闭在错误页面中的 nginx 版本数字，这样对于安全性是有好处的。

sendfile 可以让 sendfile()发挥作用。sendfile()可以在磁盘和 TCP socket 之间互相拷贝数据(或任意两个文件描述符)。Pre-sendfile 是传送数据之前在用户空间申请数据缓冲区。之后用 read()将数据从文件拷贝到这个缓冲区，write()将缓冲区数据写入网络。sendfile()是立即将数据从磁盘读到 OS 缓存。因为这种拷贝是在内核完成的,sendfile()要比组合 read()和 write()以及打开关闭丢弃缓冲更加有效(更多有关于 sendfile)

tcp_nopush 告诉 nginx 在一个数据包里发送所有头文件，而不一个接一个的发送

tcp_nodelay 告诉 nginx 不要缓存数据,而是一段一段的发送--当需要及时发送数据时，就应该给应用设置这个属性，这样发送一小块数据信息时就不能立即得到返回值。

```
access_log off;
error_log /var/log/nginx/error.log crit;
```

access_log 设置 nginx 是否将存储访问日志。关闭这个选项可以让读取磁盘 IO 操作更快(aka,YOLO)

error_log 告诉 nginx 只能记录严重的错误

```
keepalive_timeout 10;

client_header_timeout 10;
client_body_timeout 10;

reset_timedout_connection on;
send_timeout 10;
```

keepalive_timeout 给客户端分配 keep-alive 链接超时时间。服务器将在这个超时时间过后关闭链接。我们将它设置低些可以让 ngnix 持续工作的时间更长。

client_header_timeout 和 client_body_timeout 设置请求头和请求体(各自)的超时时间。我们也可以把这个设置低些。

reset_timeout_connection 告诉 nginx 关闭不响应的客户端连接。这将会释放那个客户端所占有的内存空间。

send_timeout 指定客户端的响应超时时间。这个设置不会用于整个转发器，而是在两次客户端读取操作之间。如果在这段时间内，客户端没有读取任何数据，nginx 就会关闭连接。

```
limit_conn_zone $binary_remote_addr zone=addr:5m;
limit_conn addr 100;
```

limit_conn_zone 设置用于保存各种 key（比如当前连接数）的共享内存的参数。5m 就是 5 兆字节，这个值应该被设置的足够大以存储（32K*5）32byte 状态或者（16K*5）64byte 状态。

limit_conn 为给定的 key 设置最大连接数。这里 key 是 addr，我们设置的值是 100，也就是说我们允许每一个 IP 地址最多同时打开有 100 个连接。

```
include /etc/nginx/mime.types;
default_type text/html;
charset UTF-8;
```

include 只是一个在当前文件中包含另一个文件内容的指令。这里我们使用它来加载稍后会用到的一系列的 MIME 类型。

default_type 设置文件使用的默认的 MIME-type。

charset 设置我们的头文件中的默认的字符集

以下两点对于性能的提升在伟大的 WebMasters StackExchange 中有解释。

```
gzip on;
gzip_disable "msie6";

# gzip_static on;
gzip_proxied any;
gzip_min_length 1000;
gzip_comp_level 4;

gzip_types text/plain text/css application/json
application/x-javascript text/xml application/xml application/xml+rss
text/javascript;
```

gzip 是告诉 nginx 采用 gzip 压缩的形式发送数据。这将会减少我们发送的数据量。

gzip_disable 为指定的客户端禁用 gzip 功能。我们设置成 IE6 或者更低版本以使我们的方案能够广泛兼容。

gzip_static 告诉 nginx 在压缩资源之前，先查找是否有预先 gzip 处理过的资源。这要求你预先压缩你的文件（在这个例子中被注释掉了），从而允许你使用最高压缩比，这样 nginx 就不用再压缩这些文件了（想要更详尽的 gzip_static 的信息，请点击这里）。

gzip_proxied 允许或者禁止压缩基于请求和响应的响应流。我们设置为 any，意味着将会压缩所有的请求。

gzip_min_length 设置对数据启用压缩的最少字节数。如果一个请求小于 1000 字节，我们最好不要压缩它，因为压缩这些小的数据会降低处理此请求的所有进程的速度。

gzip_comp_level 设置数据的压缩等级。这个等级可以是 1-9 之间的任意数值，9 是最慢但是压缩比最大的。我们设置为 4，这是一个比较折中的设置。

gzip_type 设置需要压缩的数据格式。上面例子中已经有一些了，你也可以再添加更多的格式。

```
# cache informations about file descriptors, frequently accessed files
# can boost performance, but you need to test those values
open_file_cache max=100000 inactive=20s;
open_file_cache_valid 30s;
open_file_cache_min_uses 2;
open_file_cache_errors on;

##
# Virtual Host Configs
# aka our settings for specific servers
##

include /etc/nginx/conf.d/*.conf;
include /etc/nginx/sites-enabled/*;
```

open_file_cache 打开缓存的同时也指定了缓存最大数目，以及缓存的时间。我们可以设置一个相对高的最大时间，这样我们可以在它们不活动超过 20 秒后清除掉。

open_file_cache_valid 在 open_file_cache 中指定检测正确信息的间隔时间。

open_file_cache_min_uses 定义了 open_file_cache 中指令参数不活动时间期间里最小的文件数。

open_file_cache_errors 指定了当搜索一个文件时是否缓存错误信息，也包括

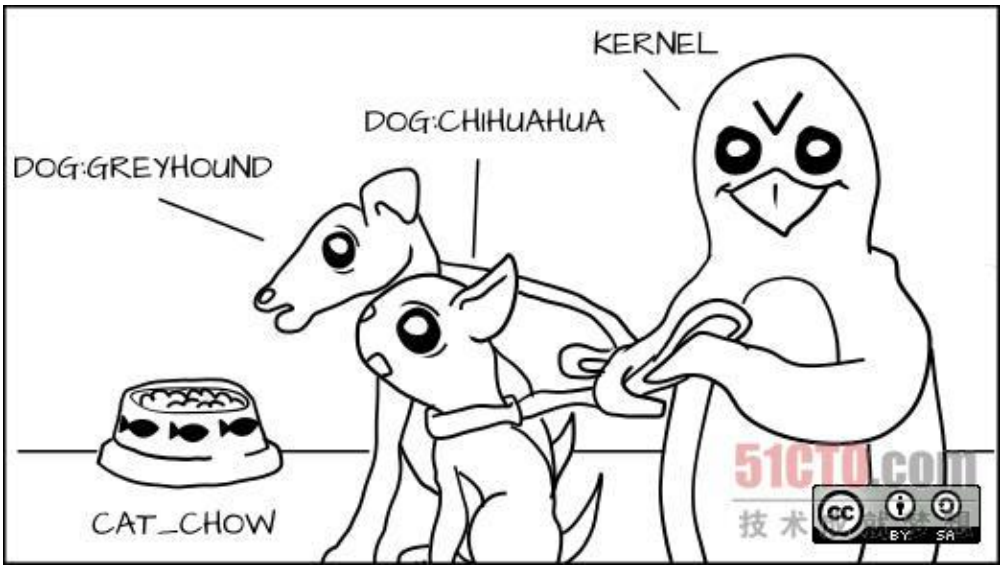
再次给配置中添加文件。我们也包括了服务器模块，这些是在不同文件中定义的。如果你的服务器模块不在这些位置，你就得修改这一行来指定正确的位置。

原文：http://www.linuxeden.com/html/news/20131121/145761.html

图文教程：SELinux 政策实施详解

SELinux 是一套标签系统。通过编写规则来控制对某个进程标签乃至对象标签的访问，这就是所谓管理政策。SELinux 内核会强制执行这些规则，有时候我们将这种方案称为强制访问控制（简称 MAC）。本文结合一些生动的趣图，对 SELinux 政策实施进行详解，带您身深入了解 SELinux。

【2013 年 11 月 21 日 51CTO 外电头条】今年我们已经迎来 SELinux 的十岁生日，回想十载发展历程真有种如梦似幻之感。SELinux 最早出现在 Fedora Core 3 当中，其后又登陆红帽企业 Linux 4。对于从未使用过 SELinux 或者希望进一步确认其基础定义的朋友们，请接着往下看。



SELinux 是一套标签系统。每个进程都拥有自己的标签，操作系统中的每个文件/目录对象都拥有自己的标签。甚至包括网络端口、设备以及潜在主机名也被分配到了标签。我们通过编写规则来控制对某个进程标签乃至对象标签的访问，这就是所谓管理政策。SELinux 内核会强制执行这些规则，有时候我们将这种方案称为强制访问控制（简称 MAC）。

对象的拥有者无权超越对象的安全属性。标准 Linux 访问控制、所有/组+权限标

记（例如 rwx）通常被称为自主访问控制（简称 DAC）。SELinux 中不存在 UID 或者文件所有权概念。所有事物都由标签机制进行控制。这意味着 SELinux 系统可以在不涉及高权限 root 进程的前提下实现设置。

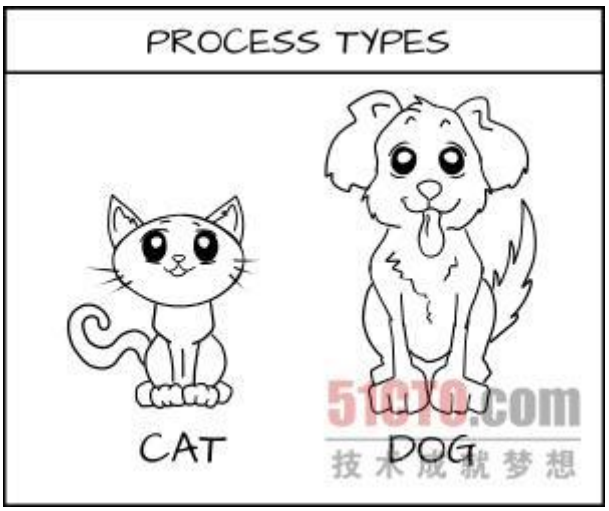
备注：SELinux 并不会取代 DAC 控制。SELinux 是一种并行执行模式，应用程序必须在同时符合 SELinux 与 DAC 的要求之后才能正常执行。这可能会让管理人员们在遇到权限被拒绝的状况时感到有些混乱。管理员权限被拒绝意味着 DAC 方面出了问题，但这肯定与 SELinux 标签无关。

- **类型强制**

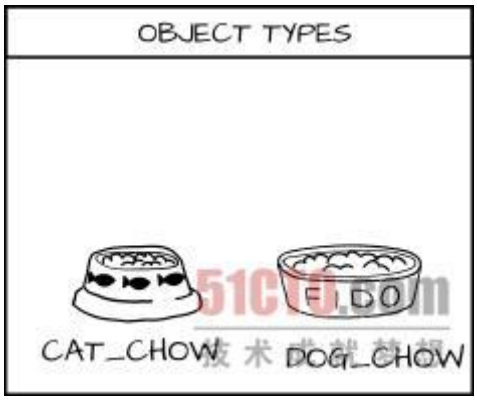
让我们进一步了解标签机制。SELinux 的主要模式或者强制手段被称为"类型强制"。这意味着我们需要根据进程的实际类型为其定义标签，而文件系统对象的标签也同样基于其类型。

比喻

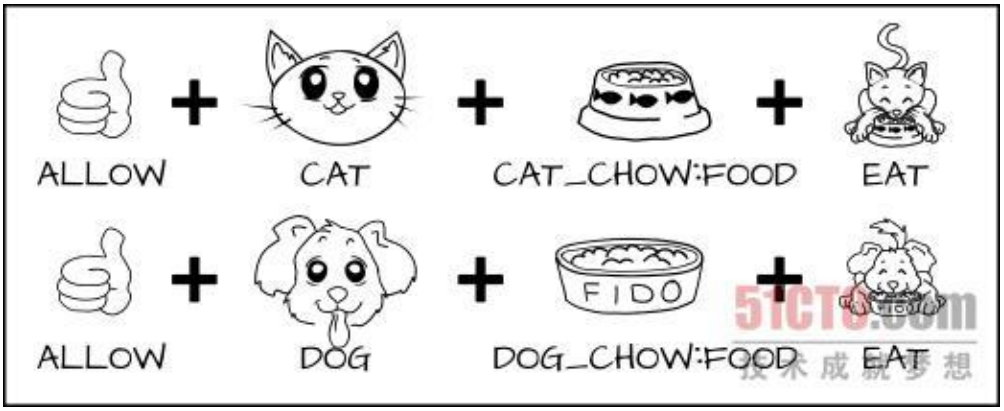
让我们假设有这样一套系统，其中需要定义的对象类型分为猫（cat）或狗（dog）。一只猫与一条狗构成进程类型。



现在我们的一类对象希望与所谓“食物”（food）进行互动。食物也需要分为不同类型，也就是猫食与狗食。

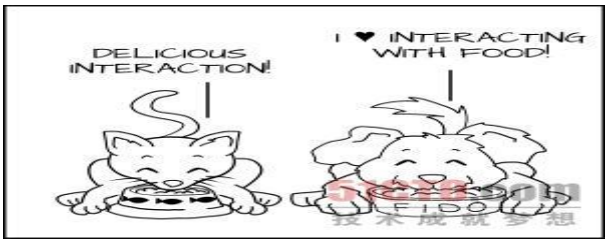


作为政策的制定者，我打算采取这样的处理方式：dog 有权吃掉 dog_chow 食物，猫则有权吃掉 cat_chow 食物。在 SELinux 中，我们会把这条规则写入政策。

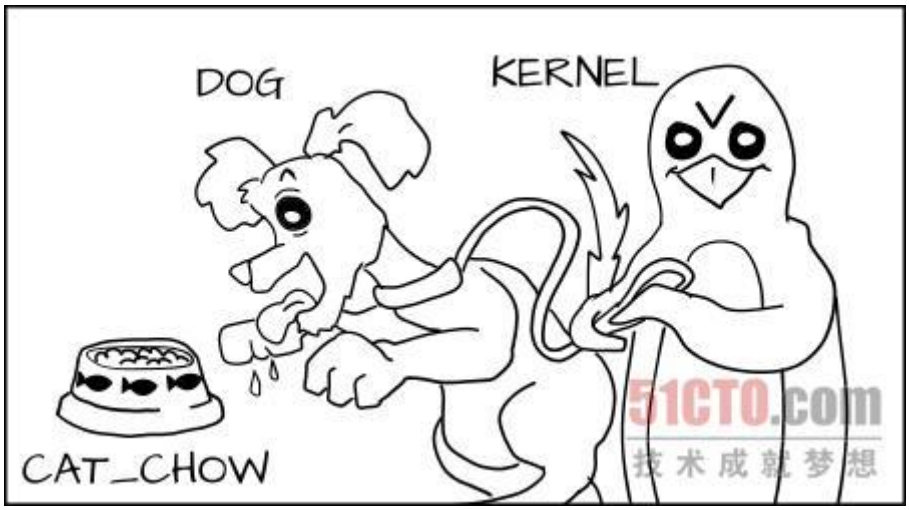


- 允许猫进程吃掉 cat_chow:food;
- 允许狗进程吃掉 dog_chow:food;

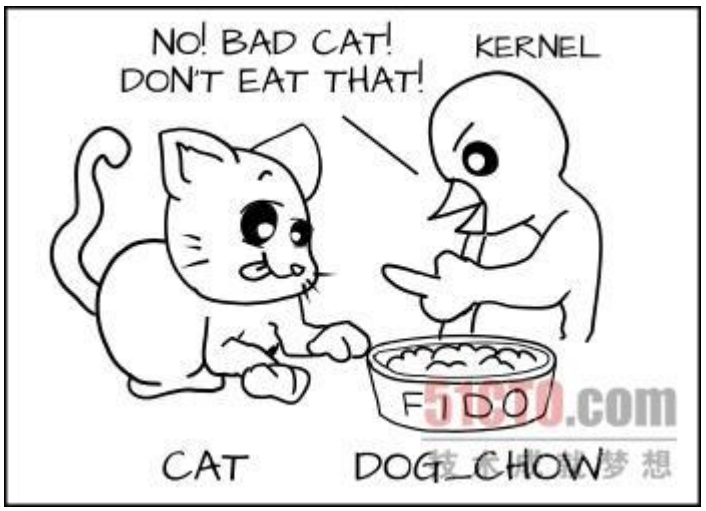
有了这些规则，系统内核将允许猫进程吃掉拥有 cat_chow 标签的食物，而狗进程则吃掉拥有 dog_chow 标签的食物。



但在默认情况下，SELinux 系统会阻止一切执行请求。这意味着如果狗类进程试图吃掉 cat_chow，内核会对此加以阻止。



与之类似，猫进程也不允许接触狗进程的食物。



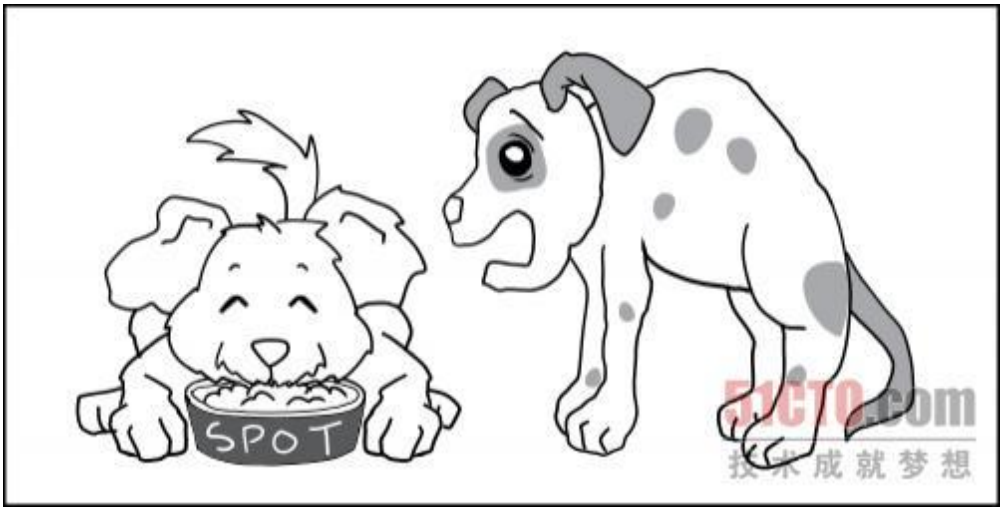
实际情况

我们将 Apache 进程标签为 httpd_t，并将 Apache 内容标签为 httpd_sys_content_t 与 httpd_sys-content_rw_t。假设我们把信用卡数据保存在一套 MySQL 数据库当中，其标签为 mysqld_data_t。如果某个 Apache 进程并侵入，黑客就能够获得 httpd_t 进程的控制权并获准读取 httpd_sys_content_t 文件的内容、向 httpd_sys_content_rw_t 当中写入数据。然而黑客仍然无法读取信用卡数据（mysqld_data_t），即使被侵入的进程以 root 权限运行。在这种情况下，SELinux 能够显著缓解由侵入活动带来的安全威胁。

- **MCS 强制**

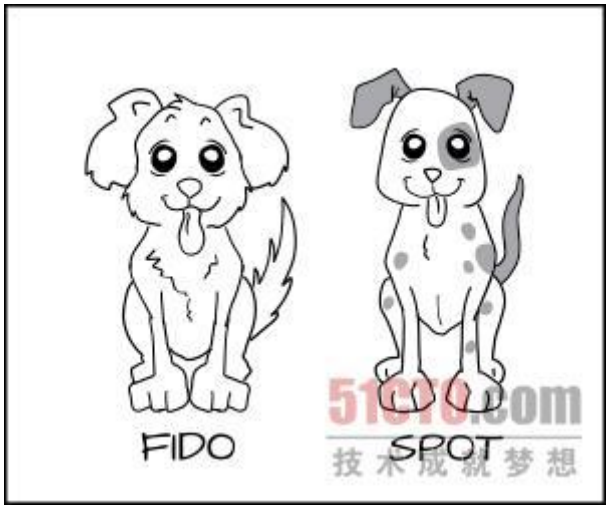
比喻

在前面提到的状况里，我们输入了狗与猫两种进程类型。不过如果狗进程又分为两种：Fido 与 Spot，我们又不希望 Fido 去动 Spot 的 dog_chow，情况又会发生怎样的变化？

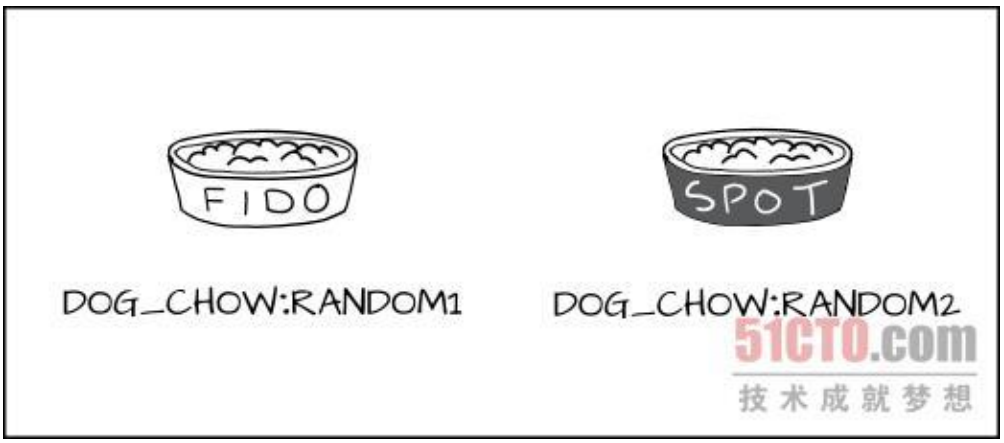


一种解决方案当然是创建大量新类型，例如 Fido_dog 与 Fido_dog_chow。不过这种作法很快就将失去可行性，因为所有狗类进程都拥有近乎相同的权限。

为了解决这个难题，我们开发出一种新的强制形式，我们将其称为多类别安全（简称 MCS）。在 MCS 当中，我们为标签加入另一种组成部分，并将其应用到狗进程与 dog_chow food 当中。现在我们把狗进程分别标记为 dog:random1（代表 Fido）与 dog：random2（代表 Spot）。

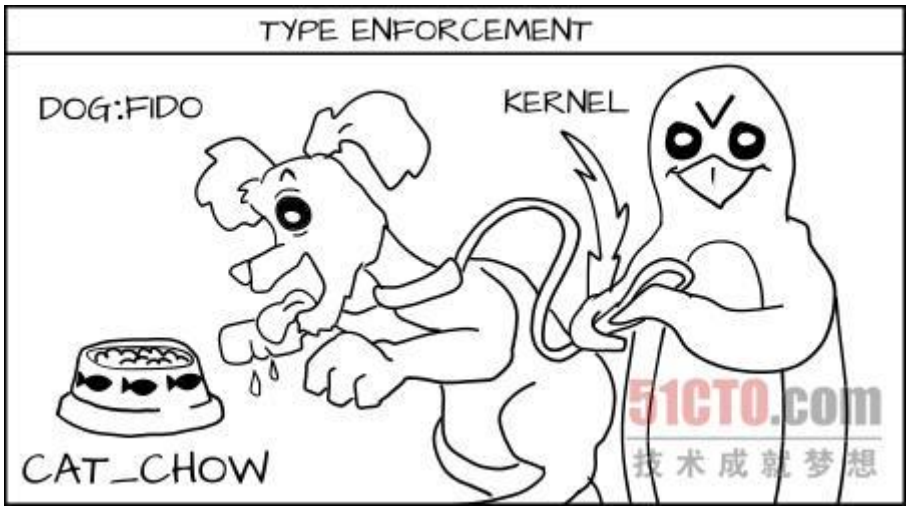


我们将狗食标记为 dog_chow:random1(Fido)与 dog_chow:random2(Spot)。



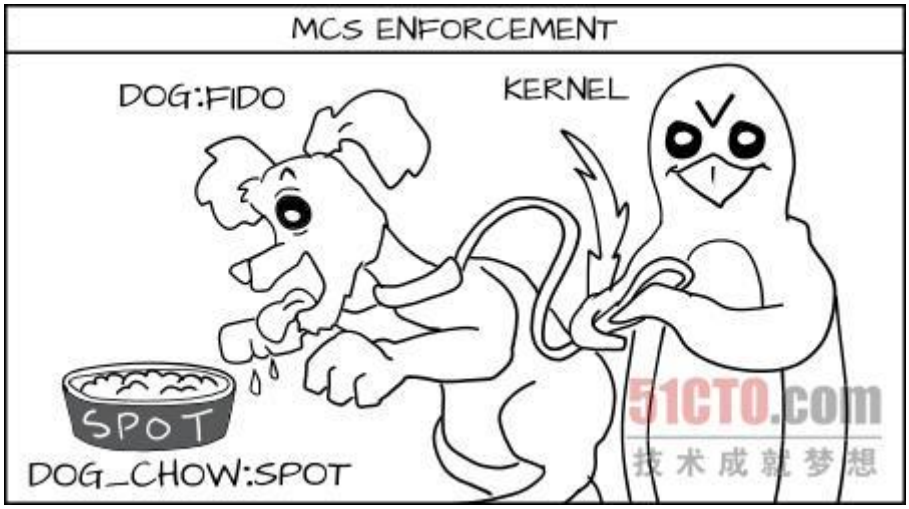
MCS 规则的内容是：如果类型强制规则没问题、随机 MCS 标签也确切匹配，那么访问才会被通过；如果二者中有一项不符合要求，访问就会被拒绝。

Fido (dog:random1)如果尝试吃掉 cat_chow:food，则会被类型强制所拒绝。



Fido (dog:random1)获准吃掉 dog_chow:random1。

Fido（dog:random1）无法吃掉 Spot 的（dog_chow:random2）食物。



真实情况

在计算机系统当中，我们通常会面对大量访问相同对象的进程，但我们希望它们彼此之间能够被区分开来。我们有时候会将此称为多租户环境。处理此类环境的最佳方案就是虚拟机系统。如果我拥有一台运行着大量虚拟机的服务器，而其中一套虚拟机已经受到黑客入侵，我肯定希望阻止黑客以此为跳板进一步扰乱其它虚拟机以及虚拟机镜像。不过在类型强制系统当中，KVM 虚拟机会被标记为 svirt_t，而镜像被标记为 svirt_image_t。我们的规则是允许 svirt_t 读取/写入/删除拥有 svirt_image_t 标签的内容。在虚拟机标签机制当中，我们不仅要采用类型强制方案，同时也要通过 MCS 对其加以区分。当虚拟机组即将启动一套虚拟机系统时，会为其选择一个随机 MCS 标签，例如 s0:c1,c2，而后将 svirt_image_t:s0:c1,c2 标签分配给该虚拟机需要管理的所有内容。最后，虚拟机会以 svirt_t:s0:c1,c2 的标记启动。这时 SELinux 内核能够控制 svirt_t:s0:c1,c2 无法向 svirt_image_t:s0:c3,c4 写入内容，而且这种控制能力在虚拟机受到黑客掌握且对方拥有 root 权限的情况下也同样有效。

我们在 OpenShift 当中也使用类似的隔离机制。每个组件（包括用户/应用程序进程）都以同样的 SELinux 类型加以运行（openshift_t）。政策定义规则、规则决定如何控制与组件类型及独特 MCS 标签相关的访问活动，从而确保不同组件之间无法交互。

感兴趣的朋友可以点击此处查看一段小视频，了解如果某个 Openshift 组件拥有

root 权限将引发怎样的状况。

- **MLS 强制**

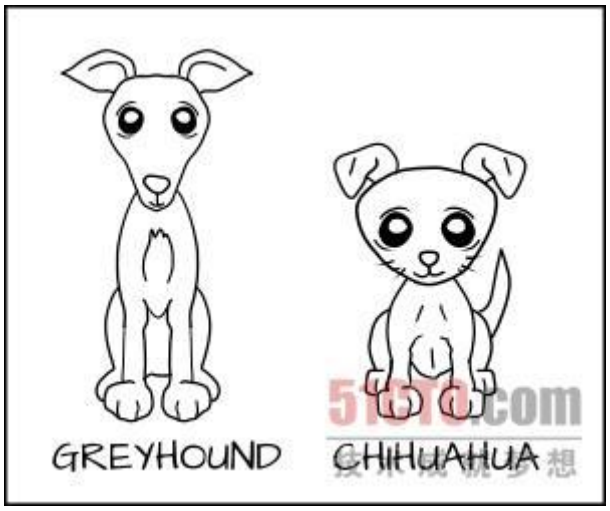
这是 SELinux 的另一种强制形式，但使用的频繁要低得多，这就是多级安全（简称 MLS）；它诞生于上世纪六十年代，主要被用于 Trusted Solaris 等受信操作系统。

其主要思路在于以数据被使用的层级为基础进行进程控制。也就是说，secrect 进程无法读取 top secret 数据。

MLS 与 MCS 非常相似，但它为强制机制添加了"支配"这一概念。MCS 标签必须在完全匹配的情况下才能通过，但一条 MLS 标签可以支配另一条 MLS 标签从而获得访问许可。

比喻

现在我们不再讨论不同的狗只，而将着眼点放在不同的狗类品种。举个例子，现在我们面对的是一只灰狗与一只吉娃娃。



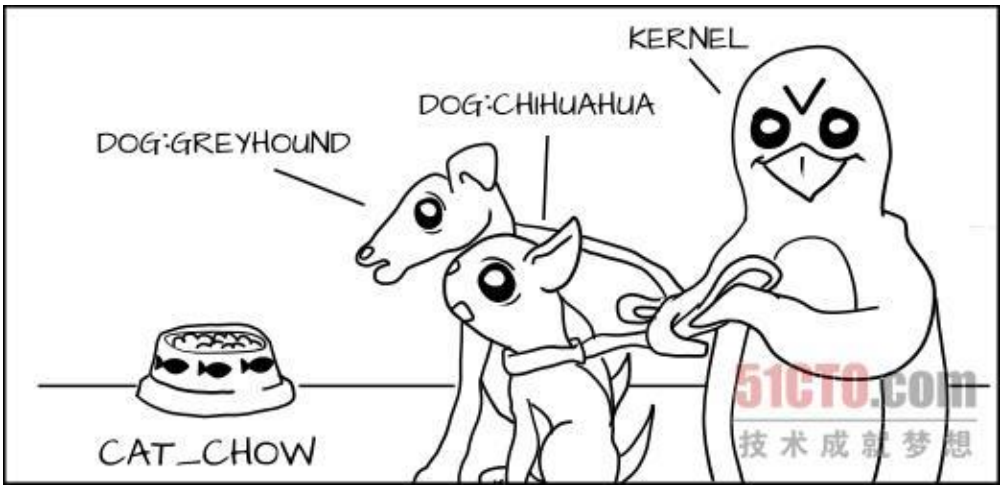
我们允许灰狗吃任何一种狗食，但吉娃娃则无法吃下灰狗的狗食。

我们为灰狗设定的标签为 dog:Greyhound，它的狗食则为 dog_chow:Greyhound；而吉娃娃的标签为 dog:Chihuahua，它的食物标签为 dog_chow:Chihuahua。

在 MLS 政策之下，我们让 MLS 灰狗标签对吉娃娃标签拥有支配权。这意味着 dog:Greyhound 可以吃掉 dog_chow:Greyhound 与 dog_chow:Chihuahua。

但 dog:Chihuahua 不允许吃掉 dog_chow:Greyhound。

当然，dog:Greyhound 与 dog:Chihuahua 仍然受到类型强制的影响而无法吃掉 cat_chow:Siamese，即使在 MLS 当中灰狗类型对暹罗猫类型拥有支配权。



真实情况

假设我拥有两台 Apache 服务器：一台以 httpd_t:Topsecret 标签运行，另一台则以 httpd_t:Secret 标签运行。如果 Apache 进程 httpd_t:Secret 遭到侵入，黑客将能够读取 httpd_sys_content_t:Secret 内容，但无法读取 httpd_sys_content_t:TopSecret 内容。

不过如果运行着 httpd_t:TopSecret 的 Apache 服务器遭到入侵，则 httpd_sys_content_t:Secrect 与 httpd_sys_content_t:TopSecret 的内容都将暴露在黑客面前。

我们将 MLS 机制用于军事环境下，即某位用户只获准查看 secret 数据，而另一位处于同一系统下的用户则可以查看 top secret 数据。

总结 SELinux 是一套强大的标签系统，通过内核对每一个单独进程进行访问控制。其中最主要的功能就是类型强制，其中规则的作用是根据进程与对象双方的标签类型来判断某个进程是否有权读取相应的对象。此外另有两种控制机制，其中 MCS 负责对同类进程进行彼此区分，而 MLS 则允许一种进程拥有指向另一种进程的支配权限。

原文 http://os.51cto.com/art/201311/418176.htm

Varnish 调优手记

最近公司做活动推广，流量暴增，后端服务器压力山大，导致用户的请求响应时间延长，客户因此抱怨声音很大。

为尽快解决问题，在安排人员不断优化后端代码的同时，考虑在 nginx 前增加 varnish 缓存层，只透传部分动态请求过去，直接减少后端服务器的压力。

在实际使用中，真正感受到了 varnish 服务器强大的威力！在不断的调优缓存命中率后，后端服务器 cpu 直接从 80%降到了 20%，再大的并发前端也可以直接消化，后端服务器表示毫无压力。有了这玩意，可以再也不用在后台写定时任务，不断重新生成静态页面了，直接丢缓存里完事！此外，varnish 还支持一种叫“神圣模式”，在后端服务器报错返回 500 的时候，varnish 还能继续优先返回过去缓存的内容，为用户屏蔽部分错误，这东东有时真算是救命稻草啊。

但同时，也趟了 n 多的坑，varnish 中的 VCL 语言太过强大和灵活，稍微运用不好就会中枪。而网上公开的大多数 varnish 配置文件都是一大抄，根本无法直接用于生产。在研究了几天，翻阅了大量各种资料后，才总算把遇到的问题都解决了。

现将调优心得记录如下：

一、介绍

Varnish 是一种专业的网站缓存软件（其实就是带缓存的反向代理服务），它可以把整个 HTTP 响应内容缓存到内存或文件中，从而提高 Web 服务器的响应速度。

Varnish 内置强大的 VCL（Varnish Configuration Language）配置语言，允许通过各种条件判断来灵活调整缓存策略。在程序启动时，varnish 就把 VCL

转换成二进制代码，因此性能非常高。

二、安装

epel 源里也有 varnish，但是却 2.x 版本的。

因为 varnish 3.0 的配置文件与 2.x 的存在很大不同，因此 varnish 团队不能再更新 epel 里的软件源。如果你想安装最新版本，推荐使用 rpm 方式。

RPM 安装

在 redhat 系服务器上可以很容易的直接通过 rpm 包安装：

view source

print?

```
wget
1 http://repo.varnish-cache.org/redhat/varnish-3.0/el6/x86_64/varnis
h/varnish-libs-3.0.4-1.el6.x86_64.rpm
wget
2 http://repo.varnish-cache.org/redhat/varnish-3.0/el6/x86_64/varni
sh/varnish-3.0.4-1.el6.x86_64.rpm
wget
3 http://repo.varnish-cache.org/redhat/varnish-3.0/el6/x86_64/varnis
h/varnish-docs-3.0.4-1.el6.x86_64.rpm
4
```

```
5 yum localinstall *.rpm
```

varnish 的安装和配置路径

view source

print?

```
1 /etc/varnish/default.vcl   #默认配置文件存文件
2  /etc/sysconfig/varnish    #服务启动参数脚本
3 /etc/init.d/varnish                #服务控制脚本
```

可以通过调整 /etc/sysconfig/varnish 配置文件中的参数来调整启动参数，设置线程池、缓存到内存还是文件等。当然如果你乐意也可以在 varnishd 后面带上启动参数手工启动服务和管理。

现在能通过服务的方式启动 varnish 了：

view source

print?

```
1 service varnish start (stop/restart)
```

将 varnish 设为开机自启动：

view source

print?

```
1 chkconfig varnish on
```

编译安装

安装依赖包

view source

print?

```
1 yum install ncurses-devel.x86_64
```

此步可选。

如果你在编译 varnish 后 bin 目录中没有发现 varnishstat、varnishtop、varnishhist 这三个程序的话，就是因为编译前没有安装与操作系统位数对应的 ncurses-devel。这些工具非常好用，因此建议先安装这个依赖包。

安装 pcre

varnish 依赖 pcre 进行 url 正则匹配。

view source

print?

```
1 cd pcre-8.12
2 ./configure --prefix=/usr/local/
3 make&&make install
```

编译

解压缩 varnish 源码包

view source

print?

```
1 wget http://repo.varnish-cache.org/source/varnish-3.0.4.tar.gz
2 cd /root
3 tar -zxvf varnish-3.0.4.tar.gz
```

```
4       cd varnish-3.0.4
5 export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
6     ./configure --prefix=/usr/local/varnish
7 make && make install
```

三、VCL 执行过程

先介绍一下 Varnish 处理请求的主要处理方法和流程

VCL 需定义几个默认的函数，在 Varnish 处理 HTTP 请求的各个阶段会回调这些函数进行处理：

- vcl_recv，请求入口，判断是否要进一步处理，还是直接转发给后端（pass）。 此过程中可以使用和请求相关的变量，例如客户端请求的 url，ip，user-agent，cookie 等，此过程中可以把不需缓存的地址，通过判断（相等、不相等、正则匹配等方法）透传给后端，例如 POST 请求，及 jsp、asp、do 等扩展名的动态内容；
- vcl_fetch，当从后端服务器获取内容后会进入此阶段，除了可以使用客户端的请求变量，还可以使用从后端获取的信息（bersp），如后端返回的头信息，具体指定此信息的缓存时间 TTL；
- vcl_miss 缓存未命中时中要做的处理
- vcl_hit 缓存命中后做的处理
- vcl_delever 发送给客户端前的处理
- vcl_pass 交给后端服务器
- vcl_hash 设置缓存的键值 key

首次请求时过程如下：

recv->hash->miss->fetch->deliver

缓存后再次请求：

recv->hash->hit->deliver（fetch 的过程没了，这就是我们要做的，把要缓存的页面保存下来）

直接交给后端 pass 的情况：

recv->hash->pass->fetch->deliver（直接从后端获取数据后发送给客户端，此时 Varnish 相当于一个中转站，只负责转发）

四、通过日志调优

安装完成后，默认的配置文件位于

/etc/varnish/default.vcl

我们可以参考缺省配置项学习 vcl 语言的使用，并进行不断的调优。

但直接修改配置，不断的重启调优效率非常低下痛苦！经过不断摸索，我发现其实 varnish 里内置了日志模块，我们可以在 defalut.vcl 最上边引用 std 库，以便输出日志：

view source

print?

```
1 import std;
```

在需要输出日志的地方，使用 std.log 即可：

view source

print?

```
1 std.log("LOG_DEBUG: URL=" + req.url);
```

这样的话，就可以通过日志了解 varnish 的工作流程，很方便的优化啦，效率何止提高十倍！

类似于你想跟踪哪些连接没有命中缓存，可以在 vcl_miss 函数中这样写：

view source

print?

```
1 sub vcl_miss {
2 td.log("url miss!!! url=" + req.url);
3 return (fetch);
4 }
```

启动 varnish 后，通过 varnishlog 工具跟踪打印出的日志

view source

print?

```
1 varnishlog -I LOG
```

五、负载均衡

Varnish 可以挂载多个后端服务器，并进行权重、轮询，将请求转发到后端节点上，以达到避免单点的问题。

举例如下：

view source

print?

```
01 backend web1 {
```

```
    .host = "172.16.2.31";
    .port = "80";
     .probe = {
          .url = "/";
         .interval = 10s;
         .timeout = 2s;
          .window = 3;
         .threshold = 3;
             }
       }
backend web2 {
    .host = "172.16.2.32";
       .port = "80";
         .probe = {
                .url = "/";
                .interval = 10s;
                 .timeout = 2s;
                  .window = 3;
                .threshold = 3;
```

```
21          }
22      }
23          # 定义负载均衡组
24  director webgroup random {
25                  {
26                  .backend = web1;
27                  .weight = 1;
28              }
29                  {
30                  .backend = web2;
31                  .weight = 1;
32              }
33  }
```

其中，在 backend 中添加 probe 选项，将可以对后端节点进行健康检查。如果后端节点无法访问，将会自动摘除掉该节点，直到这个节点恢复。

需要注意 window 和 threshold 两个参数。当有后端服务器不可达时，varnish 会时不时的报 503 错误。网上查出的资料都是改线程组什么的，经测试完全无效。后来发现，只要将 window 和 threshold 两个参数的值设成一样的，503 现象就再没有发生了。

六、优雅模式和神圣模式

Grace mode

如果后端需要很长时间来生成一个对象，这里有一个线程堆积的风险。为了避免这 种情况，你可以使用 Grace。他可以让 varnish 提供一个存在的版本，然后从后端生成新 的目标版本。当同时有多个请求过来的时候，varnish 只发送一个请求到后端服务器，在

set beresp.grace = 30m;

时间内复制旧的请求结果给客户端。

Saint mode

有时候,服务器很古怪,他们发出随机错误,您需要通知 varnish 使用更加优雅的方式处理 它,这种方式叫神圣模式(saint mode)。Saint mode 允许您抛弃一个后端服务器或者另一个尝试的后端服务器或者 cache 中服务陈旧的内容。

例如：

view source

print?

```
1           sub vcl_fetch {
2         if (beresp.status == 500) {
3                   set beresp.saintmode = 10s;
```

```
4                   return (restart);
5     }
6
7           set beresp.grace = 5m;
8                 }
```

七、完整示例

view source

print?

```
001 import std;
002
003           backend web1 {
004           .host = "172.16.2.31";
005           .port = "80";
006            .probe = {
007                     .url = "/";
008                   .interval = 10s;
009                   .timeout = 2s;
010                    .window = 3;
```

```
011                     .threshold = 3;
012                 }
013         }
014 backend web2 {
015             .host = "172.16.2.32";
016             .port = "80";
017             .probe = {
018                     .url = "/";
019                     .interval = 10s;
020                     .timeout = 2s;
021                     .window = 3;
022                     .threshold = 3;
023             }
024     }
025         # 定义负载均衡组
026 director webgroup random {
027                 {
028                     .backend = web1;
029                     .weight = 1;
```

```
030             }
031             {
032                 .backend = web2;
033                 .weight = 1;
034             }
035 }
036
037 # 允许刷新缓存的 ip
038  acl purgeAllow {
039             "localhost";
040             "172.16.2.5";
041 }
042
043     sub vcl_recv {
044         # 刷新缓存设置
045         if (req.request == "PURGE") {
046                     #判断是否允许 ip
047                 if (!client.ip ~ purgeAllow) {
048                             error 405 "Not allowed.";
```

```
049             }
050             #去缓存中查找
051             return (lookup);
052         }
053
054     std.log("LOG_DEBUG: URL=" + req.url);
055
056     set req.backend = webgroup;
057     if (req.request != "GET" && req.request != "HEAD" &&
    req.request != "PUT" && req.request != "POST" && req.request !=
    "TRACE" && req.request != "OPTIONS" && req.request != "DELETE")
    {
058         /* Non-RFC2616 or CONNECT which is weird. */
059         return (pipe);
060     }
061
062     # 只缓存 GET 和 HEAD 请求
063     if (req.request != "GET" && req.request != "HEAD") {
064         std.log("LOG_DEBUG: req.request not get!    " +
    req.request );
```

```
065                 return(pass);
066             }
067         # http 认证的页面也 pass
068     if (req.http.Authorization) {
069                 std.log("LOG_DEBUG: req is authorization !");
070                     return (pass);
071             }
072         if (req.http.Cache-Control ~ "no-cache") {
073                 std.log("LOG_DEBUG: req is no-cache");
074                     return (pass);
075             }
076         # 忽略 admin、verify、servlet 目录，以.jsp 和.do 结尾以及带有?的 URL,直接从后端服务器读取内容
077         if (req.url ~ "^/admin" || req.url ~ "^/verify/" || req.url ~ "^/servlet/" || req.url ~ "\.(jsp|php|do)($|\?)") {
078                 std.log("url is admin or servlet or jsp|php|do, pass.");
079                 return (pass);
080             }
081
```

```
082         # 只缓存指定扩展名的请求，并去除 cookie
0                   if (req.url ~
8 "^/[^?]+\.(jpeg|jpg|png|gif|bmp|tif|tiff|ico|wmf|js|css|ejs|swf|
3 txt|zip|exe|html|htm)(\?.*|)$") {
                        std.log("*** url is
08 jpeg|jpg|png|gif|ico|js|css|txt|zip|exe|html|htm set cached!
4 ***");
085                     unset req.http.cookie;
086                 # 规范请求头,Accept-Encoding 只保留必要的内容
087                 if (req.http.Accept-Encoding) {
                        if (req.url ~
088 "\.(jpg|png|gif|jpeg)(\?.*|)$") {
                            remove
089 req.http.Accept-Encoding;
                    } elsif (req.http.Accept-Encoding ~
090 "gzip") {
                        set req.http.Accept-Encoding =
091 "gzip";
                    } elsif (req.http.Accept-Encoding ~
092 "deflate") {
```

```
093                                 set req.http.Accept-Encoding =
                                        "deflate";
094                                 } else {
095                                 remove req.http.Accept-Encoding;
096                                 }
097                         }
098                         return(lookup);
099                         } else {
100                         std.log("url is not cached!");
101                         return (pass);
102                         }
103 }
104
105                         sub vcl_hit {
106                 if (req.request == "PURGE") {
107                         set obj.ttl = 0s;
108                         error 200 "Purged.";
109                 }
110                 return (deliver);
```

```
111 }
112
113                             sub vcl_miss {
114         std.log("################## cache miss ###################
                                url=" + req.url);
115         if (req.request == "PURGE") {
116                             purge;
117                 error 200 "Purged.";
118                 }
119 }
120
121                     sub vcl_fetch {
122         # 如果后端服务器返回错误，则进入 saintmode
          if (beresp.status == 500 || beresp.status == 501 ||
123 beresp.status == 502 || beresp.status == 503 || beresp.status ==
                                504) {
124                 std.log("beresp.status error!!! beresp.status="
                                + beresp.status);
125                 set req.http.host = "status";
126                 set beresp.saintmode = 20s;
```

```
127                 return (restart);
128             }
129                 # 如果后端静止缓存，则跳过
            if (beresp.http.Pragma ~ "no-cache" ||
130         beresp.http.Cache-Control ~ "no-cache" ||
            beresp.http.Cache-Control ~ "private") {
                std.log("not allow
131         cached!     beresp.http.Cache-Control=" +
            beresp.http.Cache-Control);
132             return (hit_for_pass);
133             }
        if (beresp.ttl <= 0s || beresp.http.Set-Cookie ||
134
            beresp.http.Vary == "*") {
            /* Mark as "Hit-For-Pass" for the next 2 minutes
135
            */
136             set beresp.ttl = 120 s;
137         return (hit_for_pass);
138             }
139
140         if (req.request == "GET" && req.url ~
```

```
"\.(css|js|ejs|html|htm)$") {
141             std.log("gzip is enable.");
142             set beresp.do_gzip = true;
143             set beresp.ttl = 20s;
144         }
145
        if (req.request == "GET" && req.url ~
14  "^/[^?]+\.(jpeg|jpg|png|gif|bmp|tif|tiff|ico|wmf|js|css|ejs|swf|
6   txt|zip|exe)(\?.*|)$") {
                std.log("url
147 css|js|gif|jpg|jpeg|bmp|png|tiff|tif|ico|swf|exe|zip|bmp|wmf is
                cache 5m!");
148             set beresp.ttl = 5m;
        } elseif (req.request == "GET" && req.url ~
149
                "\.(html|htm)$") {
150             set beresp.ttl = 30s;
151         } else {
152             return (hit_for_pass);
153         }
154
```

```
155         # 如果后端不健康，则先返回缓存数据 1 分钟
156         if (!req.backend.healthy) {
157             std.log("eq.backend not healthy! req.grace = 1m");
158             set req.grace = 1m;
159         } else {
160             set req.grace = 30s;
161         }
162         return (deliver);
163 }
164
165  # 发送给客户端
166 sub vcl_deliver {
167         if ( obj.hits > 0 ) {
168         set resp.http.X-Cache = "has cache";
169             } else {
170         #set resp.http.X-Cache = "no cache";
171         }
172         return (deliver);
173 }
```

八、管理命令

跟随 varnish 会一起安装一些方便的调试工具，用好这些工具，对你更好的应用 varnish 有很大的帮助。

varnishncsa（以 NCSA 的格式显示日志）

通过这个命令，可以像类似于 nginx/apache 一样的显示出用户的访问日志来。

varnishlog（varnish 详细日志）

如果你想跟踪 varnish 处理每个请求时的详细处理情况，可以使用此命令。

直接使用这个命令，显示的内容非常多，通常我们可以通过一些参数，使它只显示我们关心的内容。

- -b \\只显示 varnish 和 backend server 之间的日志，当您想要优化命中率的时 候可以使用这个参数。
- -c \\和-b 差不多，不过它代表的是 varnish 和 client 端的通信。
- -i tag \\只显示某个 tag，比如“varnishlog -i SessionOpen”将只显示新会话，注意，这个地方的 tag 名字是不区分大小写的。
- -I \\通过正则表达式过滤数据，比如“varnishlog -c -i RxHeader -I Cookie”将显示所有接到到来自客户端的包含 Cookie 单词的头信息。
- -o \\聚合日志请求 ID

例如：

varnishlog -c -o /auth/login　这个命令将告诉您来自客户端（-c）的所有包含”/auth/login” 字段（-o）请求。

varnishlog -c -o ReqStart 192.168.1.100　只跟踪一个单独的 client 请求

varnishtop

您可以使用 varnishtop 确定哪些 URL 经常被透传到后端。

适当的过滤使用　-I，-i，-X　和-x 选项，它可以按照您的要求显示请求的内容，客
户端，浏览器等其他日志里的信息。

varnishtop -i rxurl　　\\您可以看到客户端请求的 url 次数。

Varnishtop -i txurl　　\\您可以看到请求后端服务器的 url 次数。

Varnishtop -i Rxheader -I Accept-Encoding \\可以看见接收到的头信息中有有多少次
包含 Accept-Encoding。

varnishstat

显示一个运行 varnishd 实例的相关统计数据。

Varnish 包含很多计数器，请求丢失率，命中率，存储信息，创建线程，删除对象等，几乎所有的操作。通过跟踪这些计数器，可以很好的了解 varnish

运行状态。

varnishadm

通过命令行，控制 varnish 服务器。可以动态的删除缓存，重新加载配置文件等。

管理端口有两种链接方式:

1,telnet 方式,可以通过 telnet 来连接管理端口.如:"telnet localhost 6082"

2,varnishadm 方式,可以通过 varnish 自带的管理程序传递命令.如:varnishadm -n vcache -T localhost:6082 help

动态清除缓存

varnishadm -S /etc/varnish/secret -T 127.0.0.1:6082 ban.url /2011111.png

其中：ban.url 后的路径一定不要带 abc.xxx.com 域名之类的，否则缓存清除不了。

清除包含某个子目录的 URL 地址：

/usr/local/varnish/bin/varnishadm -S /etc/varnish/secret -T 127.0.0.1:6082 url.purge /a/

不重启加载配置文件

登陆到管理界面

/usr/local/varnish/bin/varnishadm -S /etc/varnish/secret -T 127.0.0.1:6082

加载配置文件

vcl.load new.vcl /etc/varnish/default.vcl

编译出错的话会有提示，成功会返回 200

加载新配置文件

```
vcl.use new.vcl
```

此时新的配置文件已经生效！

原文 http://my.oschina.net/u/572653/blog/178201#OSC_h1_1

一个完整的配置

```
user www-data;
pid /var/run/nginx.pid;
worker_processes auto;
worker_rlimit_nofile 100000;

events {
    worker_connections 2048;
    multi_accept on;
    use epoll;
}

http {
    server_tokens off;
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;

    access_log off;
    error_log /var/log/nginx/error.log crit;

    keepalive_timeout 10;
    client_header_timeout 10;
    client_body_timeout 10;
    reset_timedout_connection on;
    send_timeout 10;

    limit_conn_zone $binary_remote_addr zone=addr:5m;
    limit_conn addr 100;

    include /etc/nginx/mime.types;
    default_type text/html;
    charset UTF-8;

    gzip on;
    gzip_disable "msie6";
    gzip_proxied any;
    gzip_min_length 1000;
    gzip_comp_level 6;
    gzip_types text/plain text/css application/json
application/x-javascript text/xml application/xml application/xml+rss
text/javascript;
```

```
    open_file_cache max=100000 inactive=20s;
    open_file_cache_valid 30s;
    open_file_cache_min_uses 2;
    open_file_cache_errors on;

    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/sites-enabled/*;
}
```

编辑完配置后，确认重启 nginx 使设置生效。

```
sudo service nginx restart
```

后记

就这样！你的 Web 服务器现在已经就绪，之前困扰你的众多访问者的问题来吧。这并不是加速网站的唯一途径，很快我会写更多介绍其他加速网站方法的文章的。

原文 http://my.oschina.net/u/572653/blog/178201

小 S 的困惑

小 S 是一名新员工，他和很多踌躇满志的大学毕业生一样，实习+工作，他来到了一家非常对口自己爱好的公司，来到了一支温暖的**团队 A**，这支 30 人的大团队由老员工和新员工混合组成，年龄结构复合，有男有女，有从二十几岁到四十几岁的程序员，做的视频编解码项目。整个项目组的成员都是视频编解码领域的能手或专家，最多的有 10 年的相关经验，也有几项专利，小 S 觉得这样的人应该很耐得住寂寞，有很深的造诣。

有一位导师手把手地带着他学习和进入项目，陪他一起吃饭，和他聊天，给了他公司内部通用的学习材料。于是他很快上手，最开始有一些疑惑，但是小 S 积极地去询问问题，导师和同事都很乐于帮助他，于是他进步很快。公司有一个专门帮助新员工入职和进入项目的流程计划表，小 S 严格遵照计划表上的安排行事，虽说也经常显得笨拙和死板，但它还是依靠它翻越了一个又一个困难的山头。

项目组对于代码的质量要求非常高，功能上严格测试，性能上精确控制，小 S 的每份代码都要经过至少两位同事的评审，都认可了才能提交上配置库。公司还经常组织相关的技术培训活动，甚至花钱把小 S 送到国内的一些大学培训机构去参加软件方面的培训。公司还有专门的测试团队，对于每一份内部正式发布的代码，都有完整的测试流程保证质量。对于要做什么，小 S 也是非常清楚，公司的

产品经理认真负责，写出来的设计文档一丝不苟，小 S 所在的团队拿到以后几乎不用什么沟通来确认澄清就可以开始设计开发工作。

可是渐渐地，随着小 S 熟悉了其中的流程，他开始觉得厌烦，流程极其冗长，而且工作看起来似乎太缺乏挑战，整天和视频编解码打交道，时间长了无疑枯燥乏味，缺乏动力。为了修改一个变量，需要修改、评审、提交代码，经过反复和严格的测试，可能几秒钟的修改需要十几个小时投入的测试来保证正确性。每次想做一些有意思的项目，想聊一些有趣的事情，可是同事们的话题似乎总是围着那么几件事情转：房子、车子、孩子。大家都如此敬业，按时上班，忙的时候加一会儿班，但是似乎大家并没有什么激情。

小 S 觉得他自己不能在这里继续荒废青春了，虽然他学到了很多东西，但是他的工作已经无法让他充满动力地去迎接新的一天了。他决定改变，于是他和主管提出要离开。他的第二份工作让他眼前一亮，他来到了一支新**团队 B**，一共才 5 个人，成员都在三十岁左右，清一色的男程序员，做的是通用平台开发和基础业务定制。需要做的事很杂，而项目组的员工技术和业务背景也非常复杂，有的是做底层开发做了 5、6 年，有的以前是 DBA，有的是做互联网的业务设计开发的。

面对的是陌生的领域，小 S 觉得压力颇大。经理给他分配的导师，但是看起来导师却并不积极和热情，小 S 问什么，导师就回答什么，绝不多扯。公司也没有什么通用的流程和材料供新员工学习，有的只有一个大大的资料库，不系统，也缺乏整理，小 S 只能天天在互联网上搜索来解决项目中遇到的技术问题。他本想找经理帮帮忙，可是他很快发现，整个项目组处于一个非常松散的阶段，大家做事都很独立，他便放弃了总是去打扰别人的念头。

在平时的沟通当中，小 S 发现，每个人对于自己擅长领域，确实都可谓专家，说起来眉飞色舞的；可是对于领域外的事情，即便是实际工作需要，能有兴趣，却依然缺少热情。小 S 用脚本写了一个很有意思的工具，可以改进工作效率，他很想和大家分享他的小创造，可是他发现大家的热情并不高。产品经理是团队间共享的，需要做什么不会有人来主动回答，产品经理有时会把对产品的设计想法写下来，不过小 S 不怎么看，很多情况下因为太过抽象也看不很懂。公司对于流程基本是没有什么约束，几乎不写设计文档，小 S 的代码写了也没有太多人读，倒是出了问题还得自己负责，所以他还是小心地编码和测试。

小 S 又决定离开了，这一次，他在这支团队里呆的时间更短，是因为他始终觉得没能“上道”，压力巨大不说，还不知道从何入手。我是新人，我来以后要做什么？我应该先参与哪些项目？我应该先学习哪一些内容？这些问题没有任何人回答他。他觉得很茫然，又有点怀念起了以前那支团队 A。

这一次，他加入了**团队 C**，这支团队的员工都在二十岁出头，大多数都未婚，没有孩子，看起来都充满热情和斗志。小 S 觉得自己的热情一下子被点燃了，他觉

得好长时间了，终于有了可以大干一场的感觉。项目组做的是一个信息挖掘处理的项目，这方面本身在国内行家里手就不多，再加上团队的成员普遍比较年轻，小 S 觉得大家都还处在一个学习进步为主的阶段，大家经验都不丰富。

小 S 进项目组以后，没有人给他指派导师，大家看起来都很忙，而且似乎都没有什么下班的概念，很多人一忙就到九、十点钟才回家。但是对于小 S 的提问，大家都很有热情和兴趣来回答。在这个项目组中，系统的学习材料就更少了，小 S 从网上搜了一些英文材料，买了一些相关的书来看，依然觉得缺少别人的指点，有一种缺乏头绪的感觉。对于项目上的事情，小 S 读了产品经理的文档以后想，用户的问题已经清楚了，要做的目的确实成形了，但是要怎么去做呢？还是缺乏思路。

对于代码上面，其实项目组没有什么严格的制度来保证，但是大家讨论代码层面的设计和实现特别得有热情。经常为了一个设计上的问题，为了选择一个优雅的、易于扩展的实现方案，而争论得不可开交。虽说经常觉得同事有些小题大做，小 S 还是认为自己依然学到了不少编码和设计上的知识，也看到了邋遢程序员们追求完美的另一面。整个公司只有几个测试人员，基本上只有产品成型了以后他们才能介入，于是小 S 有时候觉得心虚，对于自己写的实现，离看到让人放心的用户效果还差得很远。

虽说已经是第三个团队了，小 S 还是想离职了，这次的原因呢，是他觉得大家都在拼命地干活，努力地学习，但是他依然非常绝望，观察了几个月，他认为这支团队根本无法就把产品做成。他们就像无头苍蝇一样，忙碌却没有方向感，没有人可以带着他们一起把东西做起来。他也不确信在这支团队里面，他自己又可以学到些什么。

小 S 现在觉得很困惑，难道找一个适合的团队就那么难吗？你又会选择哪一支团队呢？

原文 **http://www.raychase.net/1658**

工作心得总结

前言

这篇总结是我在实际工作中的一些心得体会。主要是我在工作中犯的错误然后进行总结，也是对自己的警示。我在这里先抛出一个观点：**技术能力不等同于工作能力，只能说技术能力是工作能力的一部分，在公司里会发现有些技术不错的程序员并不得志，有些技术不如他的反而得到晋升**。技术能力是工具，是一把刀，是一项很重要的技能，但是如何用好他就看每个人的功力。如果你有

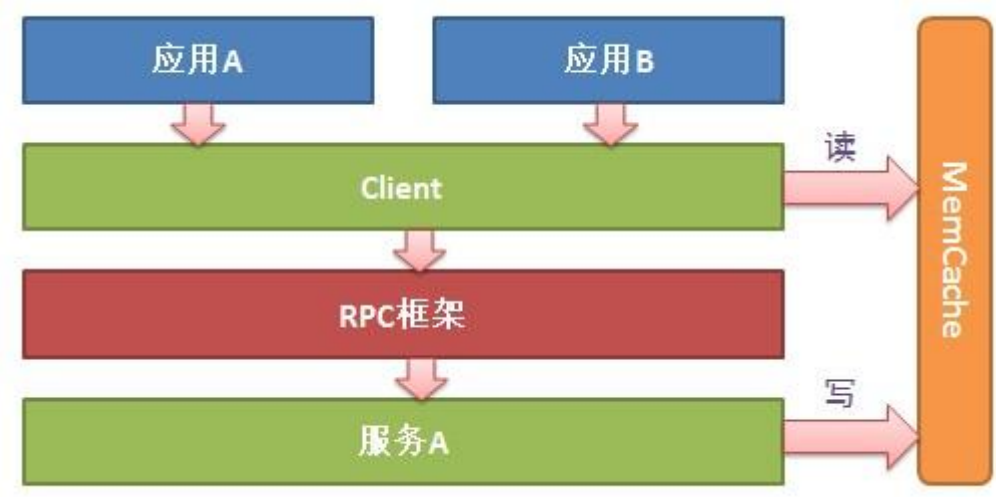
杀牛的本领，每天杀鸡有什么用，所以就是如何想方设法的把那项技能给用出来。下面的一些案例可能会有一些体会。

案例一

事件的背景是我在一个小组周会上进行了一个项目经验的分享，准备上也有些仓促，大概两三个小时写了一个简单的 PPT。讲完后就被主管批。他说：“**在用语言描述项目的时候一定要用技术性的语言进行分析：你为什么做这个项目**”。对于这句话我想很多人都不明白什么意思。这里的关键词是"**技术性的语言**"这六个字。这里我举一个我在 PPT 中描述的语言大家就会明白问题出在哪里："之前大部分依赖于数据库，对于数据库的压力**相对较大**。目前在 DB 前面用缓存挡了一层，对数据库的压力减少许多。"这里注意一下我在上面一句话中标红的那几个词。这种词是严禁在项目总结中出现，什么叫相对较大，什么叫减少许多，一切都要以**具体的数据说话**。相对较大之前的数据性能情况是多少，数据拿出来。你做了改动之后具体的数据是多少，拿出来。这前前后做一个对比，很容易就得出你做这件事的意义是什么。在这个项目中你具体做了哪些改进，而不是简简的说加了一层缓存，这样谁也不明白，你的缓存加在哪里了，是怎么实现的没有说。我刚才的问题一说出来大家都明白，具体实施的时候很多人都会犯这样的毛病。

案例二

这个案例是上面那个案例接下来的一件事。我对于一个具体的实现发起了一个讨论。先上一张图：



在周会上分享的原图我做了一些修改，这不影响接下来的分析。可能这里很多人不明白我的图。这里我先解释一下：**这其实就是一个把缓存放到服务端还是放到客户端的问题**。大家都知道远程通信都需要走网络，有网络 IO 的开销。我上面所标识的 Memcache 可以想像成集中式缓存(比如说 **Memcached**，下面我都以 **Memcached** 来写，这样可以和本地缓存区分开)，**Memcached** 通信也需要走网络 IO。我们原先的设计就是把读写 **Memcached** 放到**服务 A**，这样就会至少产生两次 IO，如果缓存失效，走 DB 的话最多会有三次 IO，**但是在应用端加一个本地缓存**。我的观点是把本地缓存去掉，然后把读 **Memcached** 放到客户端，这样大部分情况下只有走缓存这样一次 IO，对服务器的压力也减少很多。当然在会上我的观点并没有表达的很清楚。另外我这里有一个错误的观点就是**我觉得应该去掉本地缓存去掉**。我认为在这里本地缓存是不需要的。这样一种观点首先抛出来是毫无意义的，因为要不要本地缓存到底取决于具体的应用场景，而不能简单的下结论，这个问题我并没有全方位的去想。(关于性能方面可以参考《性能调优思考》)

其实在案例二中我所犯的错误一案例一中犯的错误有些类似，如果说把读 memcached 移到 Client 端去做还有些说服力，那么去掉本地缓存就毫无说服力，因为我没有任何的数据根据，只是根据一些臆测来进行。另外在整个讨论中我的思路混乱，这也是没有想清楚导致的。所以能不能全方位的思考一个问题也是一个人重要的能力。

案例三

这是一个项目中的一个案例，在提测的过程中竟然发现主功能有严重的 bug。这样的 bug 被测试发现确实非常惭愧，我把自己骂了好几遍。可能每个人都会为自己辩解，谁写代码没有问题。但是我在这里说一下我自己的体会：一般来讲写代码“**一遍率**（PS：整个逻辑盲写，不做测试）”比较高的同学往往自信心比较高，因为他对自己的代码有信心。而经常写出来代码有问题的程序员可能会**心虚**，即使你后面不管是自测还是靠测试把问题测出来，测出的 bug 越多，对于自己的打击越大。特别是一些严重依赖于开发质量的项目，这样会承受比较大的心理压力。后果是什么？有一点小的改动就会畏首畏尾，不敢改。但是真正要做到细致，以我个人的体会来看，确实很难。

另外一个就是千万在写代码之前把整个的逻辑细细的想清楚，磨刀不误砍柴工，真理。因为前期没做好的后果就是后面一直在改代码。这样浪费了更多的时间。其实这是一种**思维的转变**，很多人也包括我也认同一种观点：**代码是写出来的，即使前期想的再清楚，也会有遗漏**。但是在工作中这是一种不太好的实践。要慢慢的学会在前期做更多的工作，后期少的改动。这是一种功力，真的很

考验人。对于已经习惯这种思维的人可能不太难。但是如果习惯了在写代码中思考的程序员来说一定要力求改变，在这里也是在警告我自己。

这里简单的说一下为什么？道理很简单，如果你是在写代码的时候进行思考说明是你喜欢**发现问题解决问题**的方式，这是一种**被动的思维方式**。这种思维方式可能做一个程序员不会犯太大的错误，至多自己多加一些班。但是如果是一个项目的 owner，这样极有可能犯重大错误，整个项目到后期发现方案不可行，这是要命的。**千万不要觉得这紧紧是一种工作方式的问题，这是思维方式的问题**。要慢慢的锻炼自己在前期思维能力，就是主动思考，主动发现问题，这样才可能把项目风险掌握在自己的手中。项目实践有一句话：“**有可能发生但是没有发生的问题叫风险，如果问题已经发生，那就是真的问题**”。

改变思维方式真的很难，**要打破重来很痛苦**，绝不会在我这里写出来这么简单，所以为什么我觉得**成功学**看的热血沸腾，发现自己一去做完全是两回事。一个简单的习惯都很难改变，何况是对于一种已经几十年的思维习惯。这里我举一种思维实践，仅供参考。脑子里想一个问题，反复的想，把它想的非常透彻，然后把这个问题抛出来，看看大家都对这个问题的看法，再比对自己有哪些遗漏。这一方面是思维的过程，另一方面也算是经验积累的过程。因为很多问题想多了考虑的面自然就会丰富起来。

总结

上面讲了三个案例对于我在工作中的一些心得体会，这里面和技术没有太大的关系。我在前言中也说到了技术能力不等同于工作能力。这里可能很多人不认同，没有关系，观点可以求同存异。很多人可以从跳槽中获得薪资的提升，而公司内部的晋升确很难（这里不讨论公司等客观原因）。因为在面试的过程中大部分只会考一些技术问题。而在工作中更多的是一个人的综合能力，而这是简单的一些面试得不出来的。很多人尝到这种甜头就会一直依赖于这种跳槽来获得薪资的提升，然而更重要的是工作能力的提升，工作能力的体现就是绩效 KPI。公司永远看绩效，技术再好，结果不好枉然。这里又要叨一些玄话：以结果论英雄是公司的生存法则，也是个人的生存法则，过程是你自己的事情，对于公司而言只关心结果。

原文 http://blog.csdn.net/luohuacanyue/article/details/16367681

腾讯的 5 个月—-2013 年实习

上周五刚刚结束了 5 个月的实习，写下这篇文章记录下我这 5 个月的回忆。

- 拿到实习 Offer

今年 5 月的实习招聘中，很幸运拿到 Tencent 的终端开发(Android)实习 Offer。面试这种东西，实力最重要，但状态运气也是不能少的，在我身边，有实力但都没能拿到实习 Offer 或者好的实习 Offer 大有人在。而我算是比较幸运的一个，顺顺利利拿下第一份大公司的实习 Offer，关键是在恰当的时刻遇上了对的面试官，只要给面试官留下好印象，都会给自己加分。在这里要谢谢两位面

试官给的机会！没有他们可能我现在就不在写这篇文章了。

- 导师

6 月 20 号带着兴奋和新鲜感来到了南通入职，按流程办完入职，领取了该领取的东西，认识了我的导师，话不多的 donnie 在这几个月中给了我很大的帮助，给我指引了一条明路，让我很快容易公司的节奏中。腾讯每一位新人（校招，实习，社招）都会分配到一位导师，导师制度行程一对一的帮助制度，让新人很快能熟悉并投入到工作中，这个真的很赞。donnie 是 C++,底层高手，各种反编译，各种注入呀！这对我之后要学习和工作也产生很大的影响。 刚开始的时候，基本什么都不会，每天都有很多问题问导师，后来发现这样子是不行的，其一不能养成依赖，其二导师手头上很多工作要做，不能因为自己耽误到他的工作。后来我有问题，尝试自己解决，真没办法解决的可以放一边的放一边，汇总起来，等到导师差不多有空的时候再几个问题一起问他。

不要因为自己的问题很简单，然后不敢去问别人。自己真的解决的不了的问题，在团队中都是必须及时提出来的，不能硬撑，不然很可能出现由于个人因素而导致出现了团队的事故，到那时候挽救就晚了。在一开始，和导师的沟通存在一些困难，是因为一些术语，概念在公司有公司的说法而我们在学校可能又是一种说法，有时候导师说的东西自己会一时半伙没反应过来是什么意思，不过我还是厚着脸皮问清楚，不清楚再问题次。有人会觉得啰嗦，会觉得你怎么这都不能理解，但是却是必要的，比较深刻的是 A 说过的一句话：别人让你做一件事的时候，你要知道要你做什么，为什么要做，要怎么样去做后才能答应人家。

-学习

刚开始进来的公司，因为我应聘的职位是 Android 开发，所以我想进来以后，就是充实 Android 开发的。但其实跟我想的差距是比较大的。

进来后导师让我学习 C++， NDK ，JNI 这一些都是我之前碰都没碰过的东西，刚开始我有点不解，为什么要学这些，学这么多有用吗？C++更是我之前比较讨厌的一门语言。后来发现，其实公司内的开发一人掌握几门语言都是很普遍的事，C++ java 外带几门脚本语言，熟悉各个平台都大有人在。在这些前辈们看来，语言都只是工具，他们是这样的，遇到一个问题，这个问题适合用什么语言，什么平台来解决，那就用什么。而对于我们还在学校中学生来说，基本上都是根据编程语言或者平台来选我们的，哈哈，这就是差距，也是作为菜鸟的我的一个奋斗目标。

有一点感觉特别深刻的，就是学习时间。到了上岗工作的时候，就会深深体会到到书到用时方恨少的精髓，白天工作，晚上回到宿舍就差不多 10 点了,而且很累，洗完澡差不多睡觉了，基本上一天都没时间给你看书，这样那些工作上需要用到但自己又不会的知识点，只能留到周末两天来看书学习。那时候才真正的是“没时间读书啊”。所以大学的时光是最幸福的时光，你可以学你想要学的，任意支配你的时间。其次平时也要多积些知识，打好扎实的基础功，虽然现在用不到，但是以后可能突然就要用到了，不怕书读得不好，就怕书读得少呀！

于是乎前两个月，我几乎周末都是没休息，学习 C++ 学习 NDK 学习 JNI，Android 源码，加快自己的脚步，争取能快点开始工作，后来我被调到四组参加了 Windows 开发，有人觉得这跨度也太大了，学的东西也太多了吧。其实至始至终我觉得可以把这些东西归结为一点，就是学习能力。

做互联网行业中，优秀的人都是有着优秀的学习能力，只有学习能力强的人，团队，公司才能在快速发展的互联网大潮中制胜。公司的开发人员其实大部分时间在循环这么一个过程：发现问题－解决问题－发现问题－解决问题，这些问题中不少都是未曾遇过的，这个时候开发人员就要找到相关的资研究，然后找

到方法，再去解决问题。

解决问题的方法也是很重要，遇到同个问题，好和坏的解决方法可能花的时间是几倍之差，同事们遇到问题就翻源码找思路，我之前遇到问题是去 google 别人的思路，这也是我的一个努力方向。在工作中慢慢形成自己的一套解决方法的套路是很重要的，这是一个长期的任务。

在学习能力，解决问题能力，沟通能力面前，之前我认为最为重要的编程语言，Android 平台这些东西显得多么渺小。公司的文化和氛围给了我学习的环境。

公司亿万用户级的项目，各种具有挑战性的任务给我们的很多的锻炼，很能挖掘一个人的潜力，塑造个人综合能力和团队协助能力。

- **氛围&文化**

腾讯实行弹性工作制度，与传统行业不同的人是，腾讯没有规定上班时间和下班时间，早上一般 10 点后才开始上班（哈哈，早上睡到 8 点多），一天的任务做得差不多就可以下班，一般人都在 7-8 点才下班，也有到 10-11 点下班的。工作上，leader 分配大任务下来，我一般都会把大任务细化，拆分成几天的工作量，在一本本子上，以 list 形式记录每天的工作，下班前会统计下今天完成任务的完成度，没完成的就往下一天推，每天都会有站立晨会，简单汇报昨天和当天的工作，并了解其他人的工作情况。

公司无论职位大小，男女老少，都已英文名或者花名互称，没人叫马总，都叫 pony。这样的文化拉近了人与人之间的距离，领导和下属平时都是打成一片，拿领导开开玩笑是家产便饭，不过敢正经事的时候，大家就都会认真起来。反正要融入这个家庭，你不能严肃，不能正经，要猥琐，要恶搞，在入职第一天的自我介绍邮件中，我写得太正式了…其实合格的自我介绍邮件是要猥琐，略黄，有点暴力!!

至于穿着，我夏天是穿拖鞋上班，还有些人穿球衣秋裤，标准程序猿着装，这也是一种文化。

部门每个星期都会有很多活动，羽毛球，篮球，游泳等体育活动，两个月差不多一次部门大聚餐或者出去郊游,上次去了四海一家,最近去南沙两天一夜游，每年还有团建活动，厦门游，台湾游，日本游，斐济岛等等，每周小组都会不定时去下馆子，最近我都吃胖了几斤了，不过 leader 说是因为馆子地沟油多，除了这些还有好多好多福利。

公司对实习生很重视，主要是培养为主，5 个月我基本用 2 个多月在学习，可以说是带薪学习哈，不同其他小公司，一些小公司目的很明确，就是要让你一进去就压榨劳动力，到最后还很可能不留用呢。

腾讯去除了传统行业公司的种种束缚和官僚，加入了新鲜活跃的的元素，注重个人的发展，给人一种轻松活跃的工作氛围，这恰恰符合了这代 80 90 后年轻人的个性，也许只有这样的公司才能有创造性的产品。

- 总结

这个 5 个月的实习，说长不长说短不短，认识了很多人，学到很多之前没接触到的东西。最重要的是对自己有了重新的认识。

1. 大学毕业只是一个开始而已，前面路漫漫。

2. 与一群牛人共事，你会发现自己很渺小，但也发现路越走越宽。

3. 走出自己的小范围，到外面看看，才能更好定位自己。

4. 做任何事沟通最后总要，事前沟通，事中沟通，事后沟通。

5. 珍惜来之不易的机会，创造更多的价值。

6. 学会感恩帮助过你的人，学着帮助别人，做到别人对你心存感激。

7. 学无止境，不怕书读不好，就怕书读少。

- **感谢**

感谢这段时间帮助我，支持我的人，这是一段美好的时光。爱你们，爱这个大家庭！ 下一年再见！

原文 http://www.chenchuangfeng.com/?p=162

【我在硅谷做码农】别再羡慕硅谷的食堂了，那是个“阴谋”！

公司不仅用美食“绑架摧毁”了你的身体，并且那该死的“让公司像家一样舒适”的理念也将我们码农的生活彻底套牢——当你外出以及在家能干的所有事情，不

论是吃饭、游戏还是健身等公司都可以帮你免费解决，甚至比自己花钱获得的服务更好的时候，你还有什么理由再踏出公司一步呢？

注：**很多人都羡慕硅谷的公司里那齐备的娱乐、健身设备，并为那些硅谷办公室里各式各样的零食、餐点口水不已。在本文中，作者却对此大吐苦水，为自己“身陷‘吃海’”不能自拔，以致生活习惯越来越“肥胖”，而愤慨不已。**

这是钛媒体在 10 月 8 日改版后，隆重推出的首个独家公司人专栏 【我在硅谷做码农】之五 ，作者为我们分享作为一个码农，在硅谷有趣、有料的见闻、职场和情感纠葛，各种工作生活感悟。 前四篇回顾（【我在硅谷做码农】一、二、三、四）。

先说点别的。你们知道吗，之前听了我一系列的吐槽之后，国内码农们的反应各异，着实有趣。

有读者客气地表示，“**洋码农你太谦虚了，说到底也是精英人才**”，不敢当不敢当；还有一位表示对如此的喋喋不休实在忍无可忍，于是寄给我一箱臭鸡蛋，快递箱子上用黑色马克笔写着“**见物如见君**”几个大字；更有好事者好心劝我，“**绝不虚言，通过吐槽发泄远不如床上运动来得更直接痛快**”。

诚如所言，相当理解。

当然，也不乏遇到志同道合之人，近日，就接到一封自称为吐槽爱好者的来信，其表示既然都是码农又志趣相投那就不客气了，于是他用了足足一百页 word 对其生活进行了全方位的吐槽，在结尾还特别指出，在国内空气污染、食品安全无药可救的大背景下，你们国外码农还是相当幸福的，“**尤其是你所在的 XX 公司，福利待遇闻名遐迩，如此这般还不知足，真够可恶的！**”

冤枉啊，绝对的冤枉！

不错，我承认，我们公司确实有各色令人应接不暇的美食，不仅如此，洗衣、洗剪吹、健身房、休息室、电玩室、体检、邮寄等福利设施和服务也一应俱全，对此，有人感叹道“**即使是上帝来到人间也会想成为硅谷码农的**”。

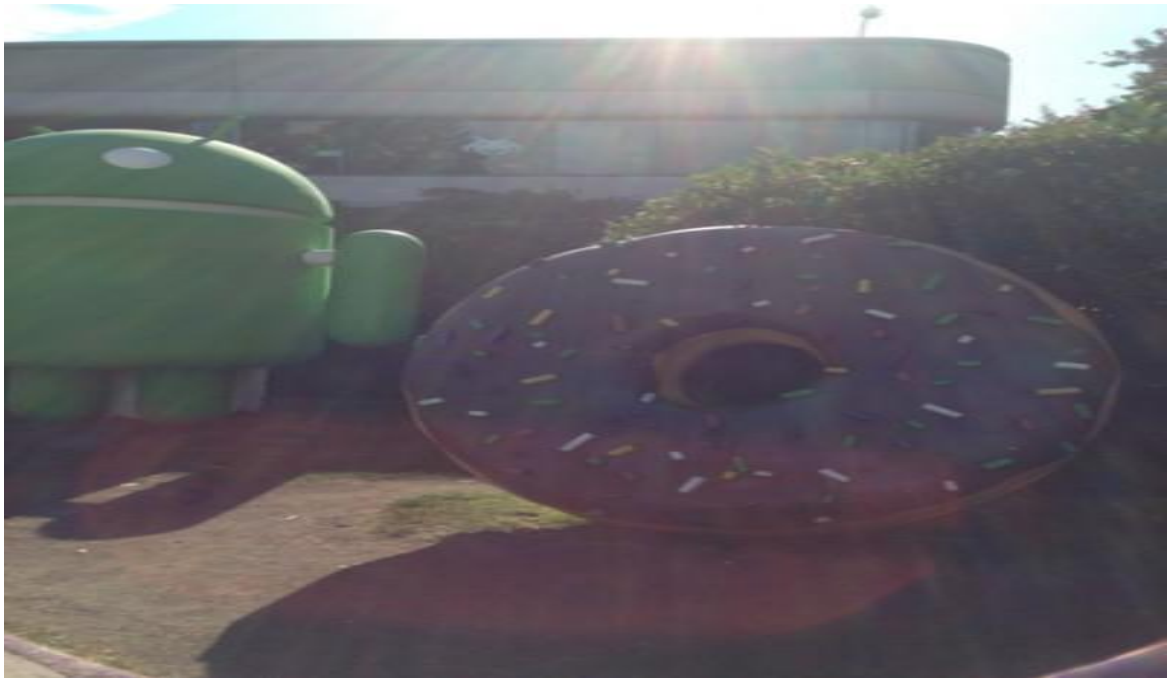
但别忘了，啥事都过犹不及。刚进公司的时候确实相当 happy，但历经了常年的吃喝玩乐之后，同公司大多数同事一样，我不仅体重飙升，而且血糖血脂血压也屡创新高。不信？近日，我们公司一名同事就愤然辞职了，理由是公司的免费食物太好，使他变胖了 20 斤，其表示“美食当前难以抗拒，我发现自己不论无聊或思考时都在吃”。

确实如此，公司不仅用触手可及、不间断供应的美食“绑架摧毁”了你的身体，让你彻底沦为脂肪的奴隶，并且那该死的“让公司像家一样舒适”的理念也将我们码农的生活彻底套牢——当你外出以及在家能干的所有事情，不论是吃饭、游戏还是健身等公司都可以帮你免费解决，甚至比自己花钱获得的服务更好的时候，你还有什么理由再踏出公司一步呢？

如此这般，你呆在公司的时间将越来越长，什么都习惯于在公司解决，那么自然，你的社交圈也将慢慢缩小至公司同事，而后，随着你脂肪和身材悄无声息的走样以及社会社交技能的进一步下降，不论是泡妹子还是交际拓展你都将优势全无，那怎么办呢？当然只能灰溜溜地躲回吃喝不愁的公司温柔乡了。

所以，明白了吧，美食和福利绝对是资本家最大的阴谋，它犹如一张巨大而无形的网，用诱人的姿态吸引你进入，然后用最高效的方式悄无声息地将你捕获，从此，从身体到心理都已产生无可救药依赖感的码农们再无挣脱的能力和勇气，于是余生都将不计时间、不计成本、死心塌地地为公司服务！呜呼~！

为求安慰，下面有必要图解一下一个码农被公司套牢的一天。



早上 9 点，当你饥肠辘辘地抵达公司，首先映入眼帘的是公司大楼外各色的操作系统标志物，它们被不怀好意地做成 cup cake、冰激凌、饼干、三明治等造型，仿佛一堆脂肪向你招手，也预示着吃货的一天又开始了！

每天进入办公楼之前，我都会路过一个巨大的甜甜圈雕塑，我忍不住咽了下口水，不顾变成旁边绿色怪物一样的水桶腰的风险迫不及待地奔向办公楼一楼的食堂。

早上 10 点，三明治、香肠、鸡蛋……早餐过后必须来一杯咖啡，不论是星

巴克还是现磨咖啡亦或是各国茶饮这里都一应俱全。不限量的 cream 和糖浆再次占领了我的胃，吃到这里我已经决定既然如此不如自暴自弃了。

中午 12 点，为了用美食绑架我们，公司无耻地搞了十五六个食堂，还分美式、中式、日式、印度、墨西哥等多种类型，分别用于搞定不同种族的码农，我最近爱上了这个日本料理食堂，虽然早餐吃太多完全不饿但也无力抵挡，算了，就随便来几盘刺身好了，大麦茶是必须的，过滤脂肪用的，这样下午就可以继续吃了，耶。

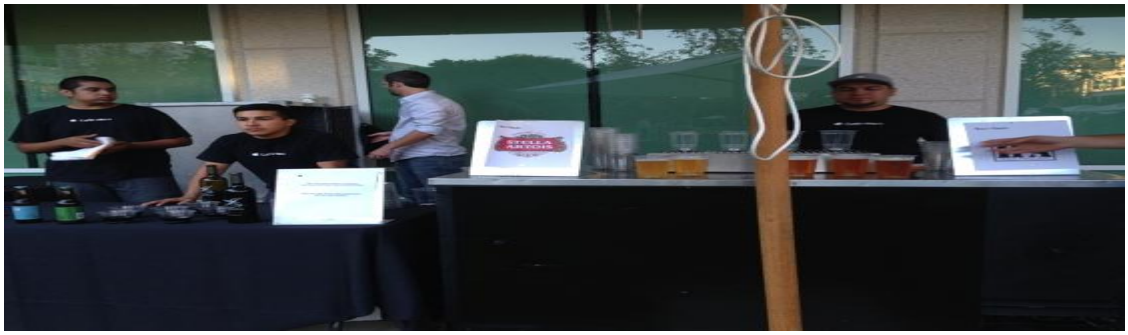
加料，加料，继续加料，什么长肉加什么，绝不吝惜！累觉不爱了。

不来几块甜点，一顿餐能算结束吗？算了，至少甜点可以安慰我被美食绑架的绝望的内心，嗯。

中午 13 点，不限量各色饮料，为每天的脂肪积累添砖加瓦，不论你好哪一口，

碳酸饮料、果汁类还是运动能量饮料这里都可以无限量满足你！如果你指望通过戒掉饮料减肥，无处不见的饮料柜绝对一举将你摧毁。

下午 15 点，各色高能量的零食犹如智能手机一般塞满你全部的碎片时间，如果你想在酒足饭饱之后让胃稍作休息，那简直太天真了，公司遍地的零食哪会让你如愿，我只能说，不是我意志力不够坚强，而是敌人实在过于强大。

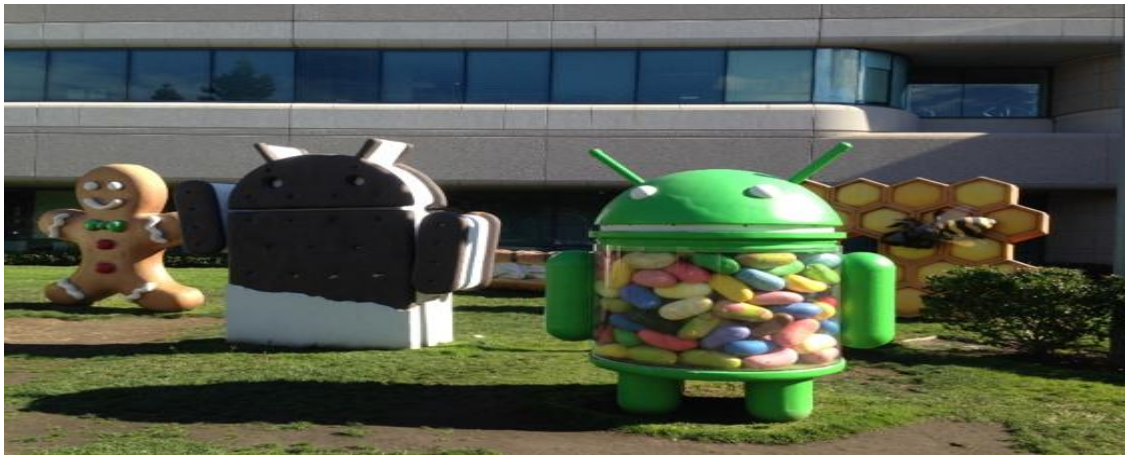


下午 16 点，公司隔三差五举行的 Beer Bash（为庆祝新品上市举办的内部 party）

为你的啤酒肚再添猛料。可劲喝哦亲！

晚上 19 点，下班后别急着走，办公室旁边就是电玩室，最新最酷最好玩哦，还免费，再说了，刚才晚饭又吃多了，还不趁机运动一下？

晚上 21 点，转眼夜幕降临，四周寂静，走出电玩室，路过一架皎洁的钢琴，反正已经这点了，接下来也无事可做，不如带着腰上的游泳圈弹奏一曲，纪念我逝去的清瘦时光。



晚上 22 点，终于可以回家了，走出办公楼，我恶狠狠地瞪了一眼这个肚子里装满糖果的肥腰绿怪，它仿佛在对我说：“明天见哦，别忘了继续吃哦，然后很快你就会拥有和我一样的好身材了哦！”F***！

怎么样，以图为证，我没说错吧，所以现在你应该深深地同情我们这群被不怀好意的资本家绑架了的码农了吧？

最近更让我郁闷的是，肥胖和生活习惯是可以传染的，由于公司允许外带食物零食饮料，老婆和儿子也开始走上了我的老路，不仅无节制地吃吃喝喝，并且越来越喜欢和我一起来公司了，于是乎，每逢下班或假期，别家都各处游山玩水，我们一家三口通常被绝望地绑在公司，过去现在和未来恐怕都将如此。

谁来解救我？！

原文 http://www.tmtpost.com/79775.html